P2 Data Structures

P2 Data Structures

Howdy

Numbers

Python has 3 data types that represent numbers:

1. Integers
2. Floats
3. Complex (much less popular than the previous 2)

Integers vs Floats

Differences

Integers are Unbounded

- Integers can be as high in magnitude as you want in Python 3 - which is not the case in Python 2
- You don't run into cases where BigInt would be required like in JS

These are valid numbers to assign to variables and use in Python

```
-2000000000000000000000000000000000000000
900000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

Floats are not Unbounded

Python has a precision limit for floats - meaning there's a maximum number of decimals it can handle

```
x = 3.11111111111111111111111111111111111111111111111111111111111111
print(x) # 3.111111111111111
```

Using Integers & Floats Together

We're getting ahead of ourselves here, but you can perform math using both integers and floats

```
# Integer and float
a = 5
b = 2.0
# Addition
result1 = a + b  # Result will be a float (7.0)
# Subtraction
result2 = a - b  # Result will be a float (3.0)
# Multiplication
result3 = a * b  # Result will be a float (10.0)
# Division
result4 = a / b  # Result will be a float (2.5)
# Exponentiation
result5 = a ** b  # Result will be a float (25.0)
# Floor Division (integer division)
result6 = a // b  # Result will be an integer (2)
# Modulus (remainder)
result7 = a % b   # Result will be a float (1.0)
```

Are not Strictly Equal

Integers and floats representing the same numeric value are not considered strictly equal

```
a = 5
b = 5.0
print(a is b) # false
c = 5
d = 5
print(a is b) # true
```

Typing #s in Unconventional Ways

Numeric Delimiter _

- When humans write down large numbers, sometimes we use delimiters like commas
  Ex. 1 million could be written as `1,000,000`
- Python allows you to use the underscore `_` as a delimiter to make large numbers more readable
  They are optional - you have the choice to just bunch all numbers together

```
x = 1000000
y = 1_000_000
print(x == y) # True
```

Integers and Leading Zeros

Leading Zeros are not permitted for integers
They are permitted in floats since lots of numbers begin in `0.`

```
x = 09 # # SyntaxError: leading zeros in decimal integer li
```

Floats and Leading Zeros

Python allows you to exclude a leading zero when representing decimals

```
x = 0.9
y = .9
print(x == y)  # True
```

Scientific Notation

- Python allows you to type out numbers in scientific notation
- Python will auto convert numbers to scientific notation if it decides it'll look better that way
    - Only happens with extremely long numbers or floats
    - May not even happen on certain python environments- the long number may just be left as is

```
x = 5e-3
print(x)   # 0.005
y = 12860000000000000000
print(y) # 1.286e+19 (depending on what python env you're u
```

DON'T WORRY ABOUT THESE MUCH
Very rarely comes into play - and these are just conversions Python does under the hood
You can perform arithmetic as if you were using regular numbers the entire time

Generating Random Numbers

You can generate random integers within a range

Random Integers

This function generates a random integer between `min` (inclusive) and `max` (inclusive)

```
import random
random_integer = random.randint(1, 3)  # Generates a random
print(random_integer) # 55
```

Random Floats (Decimals)

```
import random
random_float = random.uniform(0.0, 1.6)  # Generates a rand
print(random_float) # 1.2732697746309614
```

Rounding

Up to Nearest Integer

```
import math
math.ceil(5.1) # 6
```

Down to Nearest Integer

```
import math
math.floor(5.9) # 5
```

Standard Rounding

The `round()` method rounds up starting at 5, and down before 5 (intuitive!)

Round to 1 decimal place

```
x = round(5.75, 1) # 5.8
y = round(5.74, 1) # 5.7
```

Round to 3 decimal places

```
x = round(2.7518, 3) # 2.752
y = round(2.7512, 3) # 2.751
```

Infinity

Positive Infinity

```
float('inf')
```

Negative Infinity

```
float('-inf')
```

Save Max & Min Numbers

```
minimum = min(1, 8) # 1
maximum = max(4, 3, 99) # 99
```

< Complex: Final Number Data Type >

Data Types & Casting

List of Data Types

Not all data types will be covered in this chapter - some will have their own dedicated chapters or lessons
This chapter holds all the types that wouldn't fit elsewhere (leftovers bin, essentially)

Data Type: None

Represents Lack of a Value

Python's `None` data type is like `null` from JS

- It represents the lack of a value
- Is different than 0, false, or empty strings - since those are still values, even if they are falsy

```
x = None     # remember it is typed with a capital N
```

Use Case

There are endless use cases for the `None` data type
Oftentimes we use it for conditionals (which are covered later in the course)

```
value = None
if value is None:
    print("The value is None!")
else:
    print("The value is not None!")
```

```
The value is None!
```

Type Casting: Change Data Type of a Value

Python does not have type coercion like JS does
It'll throw errors when you try to mix and match data types when performing certain operations

```
x = "Stringified"
x + 99 # ERROR (you can't add integers to strings in Python
```

Type Cast Methods

Instead of relying on the language coercing types for you, convert them yourself

```
int()  # turns input into an integer (if a decimal is provi
float() # turns input into a float
str()  # turns input into a string
bool()  # turns input into a Boolean (good for seeing if Py
```

int()

```
a = int(1.99)   # 1
b = int(5.1)    # 5
c = int("40")   # 40
d = int(5)    # 5 (already an int)
e = int("9.0")  # Causes error. Cannot use a float string a
```

float()

```
a = float(1.99)  # 1.99 (already a float)
b = float(5)     # 5.0
c = float("40")  # 40.0
d = float("5.4")  # 5.4
```

str()

```
a = str(1.99) # "1.99"
b = str("40") # "40" (already a string)
```

bool()

```
list1 = [1, 2, 3]
is_list1_filled = bool(list)  # True
list2 = [] # empty data structures are considered Falsys in
is_list2_filled = bool(list2)  # False
```

Exercise: Age Calculator In Days

1. Ask user how old they are in years using input method

2. Change user input type from string to a number
3. Perform math with it and print the number of days old they are (as an integer)

```
years_old = input("Enter your age in years: ") # input save
days_old = int(float(years_old) * 365) # convert string to
print(f"You are {days_old} days old at the moment!")
```

Type Checking

Type Hints (not enforcement)

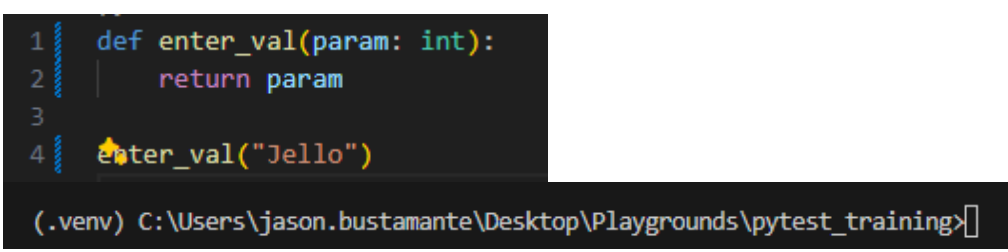
What are Type Hints?

Type hints are meant to show coders what types certain parameters should have - think of them like comments

- They don't actually enforce type checks like in TS
- Ignoring them will not cause any exceptions to be thrown when the code is ran, or cause any IDE errors to appear

```
def enter_val(param: int):
    return param
enter_val("Jello") # providing a string where an integer is
```

IDE is fine with it Python is fine with it when code is ran



What if I want Type Enforcement?

See the next lesson

Set up Type Enforcement

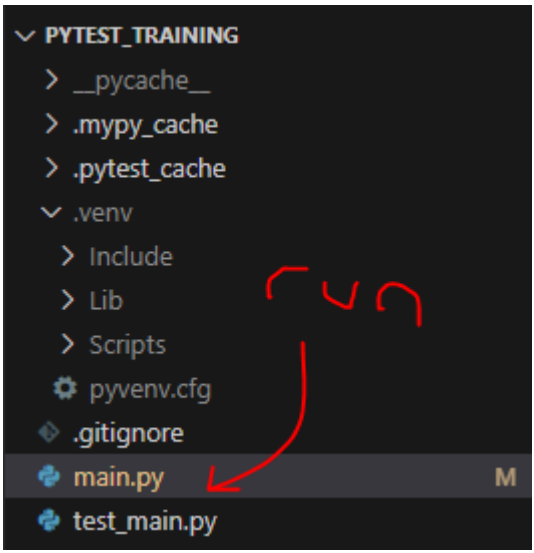
Setup: pydantic + mypy

Python is a dynamically typed language by default, so types are not enforced
If you want them to be enforced in order to code like Python is statically typed, you'll need some dependencies

```
pip install pydantic
pip install mypy
```

Once you have these, run python files using a command like this:

```
mypy main.py
```



With vs Without Type Enforcement

Here is a method whose parameter should be an integer
We're going to feed it a different type altogether and see what happens normally vs when you use mypy

```
def enter_val(param: int):
    return param
enter_val("neon noodle")
```

Run normally

- Command is `py main.py`

- Despite the fact that we ignored the param type, nothing happens (not ideal)

```
(.venv) C:\Users\jason.bustamante\Desktop\Playgrounds\pytest_training>py main.py
```

Run with mypy to enforce type checks

- Command is `mypy main.py`
- An exception is raised since the types are now enforced (good)

```
(.venv) C:\Users\jason.bustamante\Desktop\Playgrounds\pytest_training>mypy main.py
main.py:4: error: Argument 1 to "enter_val" has incompatible type "str"; expected "int"  [arg-type]
Found 1 error in 1 file (checked 1 source file)
```

Quirks with Type Enforcement

One weird quirk I found is that Booleans are considered numbers by Python Even with type enforcement, feeding a Boolean where an int should be raises no issues (strange)

Ran with `mypy main.py`

```
def enter_val(param: int):
    return param
enter_val(False) # Bools and int are the same thing type-wi
```

```
(.venv) C:\Users\jason.bustamante\Desktop\Playgrounds\pytest_training>mypy main.py
Success: no issues found in 1 source file
```

Typed Variables

You can specify types for variables when it's unclear what kind of values they'll hold

Redundant Typed Variables

When values are hardcoded, there's no point of specifying a type since mypy is smart enough to infer

```
temp: str = "storm" # no point since storm is clearly a str
```

When Typed Variables are Warranted

Often times your methods or those from 3rd party libraries need to be fed parameters of a certain type

- These are the times where defining types for your variables is required
- mypy will throw errors if the type cannot be inferred

Demonstration

COMMON PROBLEM:
When you access a value in a dict, mypy does not know what type it'll be or if it even exists

1. sample.get() may return None, which is a problem since mock_fn demands a string parameter
   None is considered an object in Python, for the record
2. Even is sample.get() does not return none, there's no guarantee what it returns will be a string

```
sample = { "apple": "one", "banana": 2 }
val = sample.get("apple") # CAUSES AN ERROR
def mock_fn(input: str) -> str:
    return input

mock_fn(val)
```

```
main.py:7: error: Argument 1 to "mock_fn" has incompatible type "object"; expected "str"  [arg-type]
Found 1 error in 1 file (checked 1 source file)
```

SOLUTION:

1. Use default return type on get() method to replace None with a desired type

2. Type cast the output from .get()

```
sample = {"apple": "one", "banana": 2}
val: str = str(sample.get("apple", ""))   # now val will alw
def mock_fn(input: str) -> str:
    return input
mock_fn(val)
```

```
C:\Users\jason.bustamante\Desktop\Playgrounds\pytest_training>mypy main.py
Success: no issues found in 1 source file
```

IN A PROFESSIONAL PROJECT
The dict probably won't be hardcoded like in our example
It'll most likely be a response from an API, or the output of another function entirely - but the solution to tackle the issue is the same

Functions with Typed Params & Returns

Simple Param & Return Type

If your functions parameter or return type is one of Python's standard types, then defining types is just like how it is in Typescript

EXAMPLE

- num parameter must be an integer
- mock_fn must return a boolean

```
def mock_fn(num: int) -> bool:
    return num > 10

mock_fn(5)
```

Typed Dict Parameter

Sometimes we must pass objects with typed key value pair values to our functions

```
from typing import TypedDict
class DictWithType(TypedDict):
    delete_storage_file: bool
    cloud_storage_id: str
def mock_method(arg: DictWithType):
    print(arg)
mock_method({ "delete_storage_file": True, "cloud_storage_i
mock_method({ "delete_storage_file": 10, "cloud_storage_id"
```

Typed Dict Return Type

Oftentimes we need to return an object with typed KVP values from our functions or endpoints

CORRECT APPROACH

```
from pydantic import BaseModel  # type: ignore
class StandardResponse(BaseModel):
    detail: str

def mock_fn() -> StandardResponse:
    return StandardResponse(detail="howdy") # THIS WORKS
```

```
mock_fn()
```

INCORRECT

```
from pydantic import BaseModel  # type: ignore
class StandardResponse(BaseModel):
    detail: str

def mock_fn() -> StandardResponse:
    return {"detail": "howdy"} # error: Incompatible return

mock_fn()
```

TypedDict with Nested Properties

You can use TypedDict from the typing module in Python to describe a dictionary with nested properties

```python
from typing import TypedDict
class Address(TypedDict):
    street: str
    city: str
    zip_code: str
class Person(TypedDict):
    name: str
    age: int
    address: Address
# Usage example
person: Person = {
    "name": "John Doe",
    "age": 30,
    "address": {
        "street": "123 Main St",
        "city": "Anytown",
        "zip_code": "12345"
    }
}
```

Literal & Enum: Define List of Acceptable Values

Function Parameter

```python
from typing import Literal
def enter_mode(param: Literal["on", "off", "deactivated"]):
    return param
enter_mode("on")
```

Combine w/ BaseModel

Oftentimes when defining request body schemas with BaseModel, we have a limited number of values we're willing to accept

SOLUTION 1: Literal

```python
from fastapi import APIRouter
from pydantic import BaseModel
from typing import Literal
router = APIRouter()
class Document(BaseModel):
    documentId: str
    title: str
    documentType: Literal["invoice", "purchase_order", "fsc
    link: str

@router.delete("/delete")
async def delete_document(data: Document):
    # ... whatever else
```

SOLUTION 2: Enum

```python
from fastapi import APIRouter
from pydantic import BaseModel
from enum import Enum
router = APIRouter()
class DocumentType(Enum):
    INVOICE = "invoice"
    PURCHASE_ORDER = "purchase_order"
    FSC_CERTIFICATE = "fsc_certificate"
    PEFC_CERTIFICATE = "pefc_certificate"
    OTHER = "other"
class Document(BaseModel):
    documentId: str
    title: str
    documentType: DocumentType
    link: str

@router.delete("/delete")
async def delete_document(data: Document):
    enum_value = data.get("documentType").get("value") # ex
    # ... whatever else
```

Access Enum Values

```python
from enum import Enum
class DocumentType(Enum):
    INVOICE = "invoice"
    PURCHASE_ORDER = "purchase_order"
    FSC_CERTIFICATE = "fsc_certificate"
    PEFC_CERTIFICATE = "pefc_certificate"
    OTHER = "other"
```

```
vari = DocumentType.FSC_CERTIFICATE.value
```

Optional Types

Syntax: With Default values

When something is optional, and you apply a default value... not providing this parameter makes it so the default value is used

```
from typing import Optional
def mock_fn(param: Optional[str] = None):
  return param
mock_fn() # WORKS
```

Without Default Value

If you make a parameter optional without a default value, you must explicitly provide a value or None
If ypu don't provide either, an exception gets raised

```
from typing import Optional
def mock_fn(param: Optional[str]):
  return param
mock_fn() # EXCEPTION THROWN
mock_fn("hi") # WORKS
mock_fn(None) # WORKS
```

List of x Type

Only lists containing integers are accepted

```
from typing import List
def mock_fn(param: List[int]):
  return param
mock_fn([1, 2, 4])
```

Union >

Disable Type Checks on 1 Line

Why Suppress Type Checks?

- There are some situations where you cannot specify types properly
  In extreme cases like this you can disable type checks on a line-per-line basis using `# type: ignore`

- Common use case:
  Some third party libraries don't have declared types, and you aren't going to go under the hood and define them yourself

Example

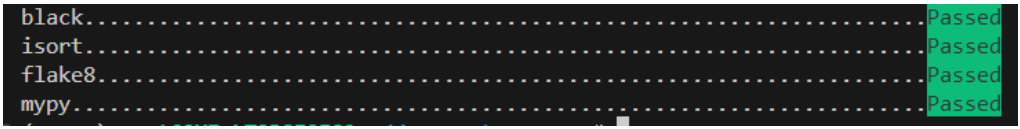
Types enabled (mypy complains about something)

```
from firebase_admin import firestore, storage
```

```
server/[redacted]_document_manager_server/routers/constants/firebase_services.py:5: error: Skipping analyzing
"firebase_admin": module is installed, but missing library stubs or py.typed marker  [import-untyped]
server/[redacted]_document_manager_server/routers/constants/firebase_services.py:5: note: See https://mypy.rea
dthedocs.io/en/stable/running_mypy.html#missing-imports
Found 1 error in 1 file (checked 4 source files)
```

Types Disabled on the problematic import line

```
from firebase_admin import firestore, storage  # type: igno
```



Use BaseModel to Apply Types

Apply types using BaseModel

Oftentimes you'll call 3rd party services that return objects of information whose types are a mystery to Python
To apply types to them, make the object an instance of a BaseModel type

```
import requests
from pydantic import BaseModel
class PlaceholderModel(BaseModel):
    userId: int
    id: int
    title: str
    body: str

api_response = requests.get("https://jsonplaceholder.typico
json_data = api_response.json() # schema unknown to python
# OPTION 1: unpack operator
```

```
typed_response_1 = PlaceholderModel(**api_response)
# OPTION 2: keyword args (use if you only want to feed some
typed_response = PlaceholderModel(
    userId=json_data['userId'],
    id=json_data['id'],
    title=json_data['title'],
    body=json_data['body']
)
```

Use BaseModel on Function Parameters

You can use a Model to impose types on method parameters

- It will work as long as the dict you supply is an instance of a Model - which we just showed you how to do in the previous subsection
- To access properties of the BaseModel instance, you must use dot notation only - bracket notation will cause an exception to be thrown

```
async def delete_file(payload: PlaceholderModel) -> None:
    user_id = payload.userId # WORKS
    title = payload["title"] # EXCEPTION THROWN
```

Util Method

Takes a json object or string then applies types to it using BaseModel schema you provide via arguments

methods.py

```
import json
```

```
from typing import Any, Dict, Type, TypeVar, Union
from pydantic import BaseModel, ValidationError
T = TypeVar("T", bound=BaseModel)
def validate_payload(payload: Union[str, dict[str, Any]], m
    try:
        if isinstance(payload, str):
            data = json.loads(payload)
        elif isinstance(payload, dict):
            data = payload
        return model(**data)
    except ValidationError as e:
        raise ValueError(f"Schema validation failed: {str(e
```

other_file.py

```
class UpdateModel(BaseModel):
    name: str
    age: int
data = requests.get("some_third_party_service")
payload = validate_payload(data, UpdateModel)
```

Ignore Certain Properties in BaseModel

You can choose to ignore all other fields provided aside from the ones you specify in the BaseModel class

```
class QueryFilter(BaseModel):
    fieldName: str
    value: str
    operation: str
    model_config = ConfigDict(extra='ignore')
```

Turn BaseModel Instance into a Dict

Standard

Use `.model_dump()` on the instance

```
metadata = MetadataModel(
            title=payload.title or file.filename or cloud_s
            upload_date=now,
            last_updated=now,
            document_type=payload.documentType,
            file_sizes_mb=[file_size],
            cloud_storage_ids=[cloud_storage_id],
            latest_cloud_storage_id=cloud_storage_id,
            file_format=file.content_type,
            entity_id=None,
            entity_type=None,
        ).model_dump()
```

To JSON format

If you want to make it JSON serializable - meaning the dict only contains types that can be directly converted to json
strings, numbers, lists, and dictionaries. Non-JSON-serializable Python objects will be excluded or converted to a JSON-compatible representation

```
from pydantic import BaseModel
class ExampleModel(BaseModel):
    name: str
    value: int
    timestamp: datetime
example = ExampleModel(name="example", value=42, timestamp=
# Using mode="json"
json_serializable_output = example.model_dump(mode="json")
print(json_serializable_output)
# Output: {'name': 'example', 'value': 42, 'timestamp': '20
# Using mode="python"
python_serializable_output = example.model_dump(mode="pytho
print(python_serializable_output)
# Output: {'name': 'example', 'value': 42, 'timestamp': dat
```

BaseModel with Nested Types

Definition

You can define a Model with nested types as follows

```
from pydantic import BaseModel
class Address(BaseModel):
    street: str
    city: str
    zip_code: str
class Person(BaseModel):
    name: str
    age: int
    address: Address
# Create an instance of Person
person = Person(
    name="John Doe",
    age=30,
    address=Address(street="123 Main St", city="Anytown", z
)
```

Setting Nested Values

You can use named parameters to declare instances of a BaseModel
If your BaseModel has nested properties, you need to reference the nested BaseModel classes

```
from pydantic import BaseModel
from typing import Optional
class ProfileContact(BaseModel):
    name: str
    email: str
class ProfileAddress(BaseModel):
    country_code: str
class BusinessProfile(BaseModel):
    business_name: str
    logo_base64: Optional[str] = None
    contact: ProfileContact
    address: ProfileAddress
business_profile_instance = BusinessProfile(
   business_name="Acme Corp",
   logo_base64="LOGO",
   contact=ProfileContact( # must reference the nested Base
       name="John Smith",
       email="john@acmecorp.com",
   ),
   address=ProfileAddress( # must reference the nested Base
       country_code="US",
   ),
)
```

Looping over BaseModel

You can run a for loop on a BaseModel instance

```
from pydantic import BaseModel
class Address(BaseModel):
    street: str
    city: str
    zip_code: str
class Person(BaseModel):
    name: str
    age: int
    address: Address
# Create an instance of Person
person = Person(
    name="John Doe",
    age=30,
    address=Address(street="123 Main St", city="Anytown", z
)
# Iterate over the Person model
for field in person:
    value = getattr(person, field)
    print(f"{field}: {value}")s
```

This script will output:

```
Copyname: John Doe
age: 30
address: address=Address(street='123 Main St', city='Anytown',
zip_code='12345')
```

If you want to access nested fields (like those in the Address model), you can add some additional logic:

```
pythonCopydef print_model(model, indent=""):
    for field in model:
        value = getattr(model, field)
        if isinstance(value, BaseModel):
            print(f"{indent}{field}:")
            print_model(value, indent + "  ")
        else:
            print(f"{indent}{field}: {value}")
# Use the function to print the entire structure
print_model(person)
```

This will produce:

```
Copyname: John Doe
age: 30
address:
  street: 123 Main St
  city: Anytown
  zip_code: 12345
```

This approach allows you to recursively process nested BaseModel instances, giving you a flexible way to work with complex data structures.

Type Conversion + Casting

Overview

Implicit Type Conversion

Python sees we assign a string value to a variable that should be an integer

- It will allow Python to convert the value to an integer automatically if it can
- If that is not possible, an exception gets thrown

```
from text_processor import get_word_count # this method ret
text = "Hello, world! Welcome to Python."
word_count: int = get_word_count(text)
```

Explicit Type Conversion

You can convert certain variables into types using the type constructor methods python has built-in

- It will convert the variable value to an integer in this case
- If it is not possible an error gets thrown

```
from text_processor import get_word_count # this method ret
text = "Hello, world! Welcome to Python."
word_count = int(get_word_count(text))
```

Cast

The cast() method does not change the type of a variable value - it just makes the type checker treat the value as the type you specify

- This is useful when you know about a value's type and the type checker does not
- Keep in mind if you use `cast()` to make the checker believe a variable is a certain type that it actually is not, no exceptions get thrown unlike the previous 2 options
  Your Python code may have runtime errors later on

```
import cast from typing
from text_processor import get_word_count # this method ret
text = "Hello, world! Welcome to Python."
word_count = cast(int, get_word_count(text))
```

```
# Past this point, word_count will still be a string, but t
```

Implicit vs Explicit

As a rule of thumb, use explicit type conversion if you're unsure Python can assign a type automatically on its own
There are some automatic conversions Python can handle, and some it cannot

Dynamic Attribute Fetch

When you access BaseModel properties with dot notation, it's not very dynamic
If you need flexibility, use the getattr method

```
field = "id"
id = getattr(basemodel_instance, field)
```

Check if BaseModel instance has a Specific Field

This is useful when writing functions that accept BaseModel instances with varying schema

```
class DocumentA(BaseModel):
    documentType: str
    content: str
```

```python
class DocumentB(BaseModel):
    title: str
    content: str

def process_document(document: Union[DocumentA, DocumentB]):
    # Method 1: Using hasattr
    if hasattr(document, 'documentType'):
        print(f"Document type is: {document.documentType}")

    # Method 2: Checking model fields
    if 'documentType' in document.model_fields:
        print(f"Document type is: {document.documentType}")
```

JSON

Convert JSON string to Python Dict

```
import json
json_string = '{"name": "John", "age": 30, "city": "New Yor
python_dict = json.loads(json_string)
print(type(python_dict)) # <class 'dict'>
```

Convert Python Dict to JSON

```
import json
data = {"key1": "value1", "key2": 123, "key3": {"nested_key
json_string = json.dumps(data)
print(type(json_string)) # <class 'str'>
```

= = = = = = = = = = STRINGS

= = = = = = = = = =

String Basics

Intro to Strings

Regular Strings

Strings are a data type in Python used to represent characters wrapped in quotations
Can use single or double quotes

```
str1 = "uno"
str2 = "www.reddit/nomadlist/8890"
print(type(str1)) # <class 'str'>
```

Using Quotation marks in Strings

- You can use single quotes inside of a string wrapped in double quotes
- Vice versa, you can use double quotes in strings surrounded by single quotes

```
str1 = "'He said what?'"
str2 = '"He said what?"'
```

If for whatever reason you need to include both, use the escape character `\` or triple quotation strings (both have future lessons)

Spaces and Tabs

Python will respect the spaces and tab-spaces inside of you string
It will not automatically cut them down to 1 space like JS does

```
str1 = "I        robot"
print(str1)
str2 = "I                        Frankenstein"
print(str2)
```

```
I       robot
I                       Frankenstein
```

WARNING:
Though this is allowed, it's more elegant to use `\t` for tabs (see escape character lesson)

len() : Get String Length

Get the length of a string using the `len()` method

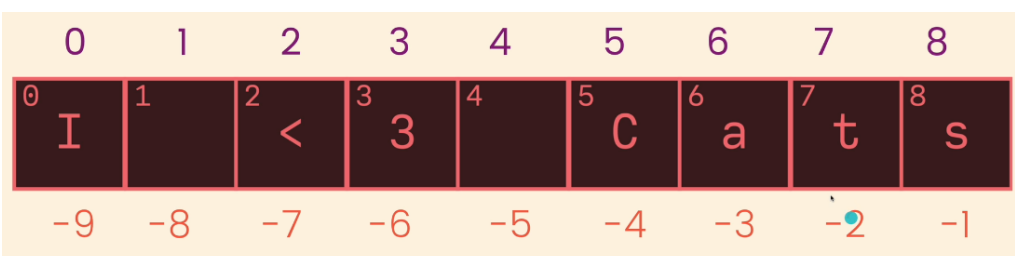
```
strLength = len("Yum Yum")
print(strLength)   # 7
```

String Operators

Repeated at JUMP119

Strings are just Arrays

Python strings are just arrays with indexed characters



Access Specific Character in a String

You can access letters with positive and negative indexes

```
unit = "second"
print(unit[0])  # get first char ( PRINTS: "s" )
print(unit[-1])  # get last char ( PRINTS: "d" )
print(unit[-2])  # get second last char ( PRINTS: "n" )
```

WARNING
If your index is out of bounds, you will get a fatal error and cause a crash

```
unit = "second"
print(unit[100])  # IndexError: string index out of range
print(unit[-100])  # IndexError: string index out of range
```

Loop through Strings

You can loop through a string just like in JS

```
for x in "ABC":
    print(x)
```

```
A
B
C
```

Generate Random UID

To generate a random uid, use the uuid library that's built into Python
Remember to convert it into a string before you use it (has a different default type when generated)

```
import uuid # built into python
id = uuid.uuid4()
string_id = str(id) # 14cd785d-393b-488e-916b-1ca100588f3a
```

```
print(type(id)) # <class 'uuid.UUID'>
print(type(string_id)) # <class 'str'> (CAST TO STRING BEFO
```

Slices

Slices with Positive Indexes

RULE WITH SECOND SLICE INDEX:
The character at the second specified index will not be included in the output
The last char to be included will be the one to the left of it

1 Index

First index only

```
msg = "Xylophone" # not affected by our slice (remains as i
sliced_msg = msg[2:]
# starts at and includes index 2, stops at end of the strin
print(sliced_msg) # lophone
```

Second Index Only

```
msg = "Xylophone" # (remains as it was originally)
sliced_msg = msg[:2]
# starts at first character, stops at char before the one a
print(sliced_msg) # Xy
```

2 Indexes

```
str[2:6]
# starts at and includes index 2
```

```
# stops at and includes index 6
```

EXAMPLE:

```
msg = "I <3 Cats" # (remains as it was originally)
sliced_msg = msg[2:6]
print(msg) # original string is not affected by you slicing
print(sliced_msg)
```

```
I <3 Cats
<3 C
```

If Second Index is too High

If the second index value you specify is too high, it won't result in any errors Python just takes a slice up to the final character in the string

```
str1 = R"C:\Users\jason\\OneDrive\Desktop" # (remains as it
print(str1[3:1000]) # Users\jason\\OneDrive\Desktop
```

Slices with Negative Indexes

> RULE WITH SECOND SLICE INDEX:
> The character at the second specified index will not be included in the output
> The last char to be included will be the one to the left of it

1 Negative Index

First Index Only

```
msg = "Elephante" # (remains as it was originally)
sliced_msg = msg[-4:] # start at and include char at index
print(sliced_msg) # ante
```

Second Index Only

```
msg = "Elephante" # (remains as it was originally)
sliced_msg = msg[:-1] # start at beginning, end at char bef
print(sliced_msg) # Elephant
```

2 Negative Indexes

With Negative Indexes

```
msg = "Elephante" # (remains as it was originally)
sliced_msg = msg[-6:-2]
# start at and include char at index -6 (p)
# finish at character to the left of char -2 (n)
# Notice we do not include character at index -2 (t)
print(sliced_msg) # phan
```

Positive then Negative Index

```
msg = "Elephante" # (remains as it was originally)
sliced_msg = msg[1: -2]
# start at and include char at index 1 (l)
# finish at character to the left of char -2 (n)
# Notice we do not include char at index -2 (t)
print(sliced_msg) # lephan
```

Negative then Positive Index

This is an odd way to slice

```
msg = "Elephante" # (remains as it was originally)
sliced_msg = msg[-8: 4]
# start at and include char at index -8 (l)
# finish at character to the left of char 4 (p)
# Notice we do not include char at index 4 (h)
print(sliced_msg) # lep
```

Slices with a Step

You can provide a 3rd value to our slicing syntax
It will allow you to skip certain characters while taking a slice

```
msg = "0123456789"
# Keep every other character when you take a slice from a s
sliced_msg1 = msg[0::2]
# Keep every 3rd char when you take a slice from index 0 to
sliced_msg2 = msg[0:8:3]
print(sliced_msg1) # 02468
print(sliced_msg2) # 036
```

Special Characters in Strings

Escape Character: \

The character `\` is used as an escape character in Python
Use it to include certain characters inside strings that would otherwise not be allowed

Common Use Cases

EXAMPLES:

```
str1 = "\"I can use double quotes in double quote encased s
str2 = '\'I can use single quotes in single quote encased s
str3 = "Word1\tWord2"
str4 = "Line 1\nLine 2"
str5 = "C:\\Users\\jason\\OneDrive\\Desktop" # backslashes
print(str1)
print(str2)
print(str3)
print(str4)
print(str5)
```

```
"I can use double quotes in double quote encased strings!"
'I can use single quotes in single quote encased strings!'
Word1   Word2
Line 1
Line 2
C:\Users\jason\OneDrive\Desktop
```

Other Use Cases

`\r` - Carriage Return
Moves the cursor to the beginning of the line without advancing to the next line. This can be used to overwrite the same line of text in a console application.

`\b` - Backspace
This moves the cursor back one space, allowing you to overwrite the last character.

`\f` - Form Feed
Advances the "paper" one page in a printer or the cursor to the next logical page in a text stream. It's not commonly used in modern text processing.

`\uxxxx` and `\Uxxxxxxxx` - Unicode Characters
Allows you to insert a Unicode character using its four-digit (`\uxxxx`) or eight-digit (`\Uxxxxxxxx`) hexadecimal representation.

`\ooo` - Octal Value
Allows you to insert a character based on its octal value.

`\xhh` - Hex Value
Allows you to insert a character based on its hexadecimal value.

Raw Strings: Fancy Characters taken as is

A string that has a leading `r or R` that is not enclosed in quotes

```
str1 = R"everything inside this string is taken as is"
str2 = r"everything inside this string is taken as is"
```

Purpose

Escape characters and their applications will be read as-is

The escape character lesson in this chapter has numerous applications of `\` which all get ignored - raw strings take what's typed in a string as is

```
str1 = "Line\nbreak"
str2 = r"Line\nbreak"
print(str1)
print(str2)
```

```
Line
break
Line\nbreak
```

Practical Use Case

When typing file paths for Windows users, you'll need backslashes `\` in your strings a lot
Though we can use escape characters to include backslashes, it is just easier to use a raw string

```
str1 = R"C:\Users\jason\OneDrive\Desktop"
print(str1) # C:\Users\jason\OneDrive\Desktop
```

Triple Quote Strings: Use Quotes & Natural Line Breaks

```
str1 = """ place string here """
str2 = ''' place string here '''
```

Purpose

Triple quotes allow you to define...

1. Strings with multiple lines
2. Strings than contain quotation marks of any kind `" or '`

Demo

We define a multi-line string with both single and double quotes used inside

```
multi_line_string = """Line1
Line2
Line3: This string can include "quotes" of both 'kinds' wit
print(multi_line_string)
```

```
Line1
Line2
Line3: This string can include "quotes" of both 'kinds' without needing to escape them
```

Notice how the printed string includes line breaks without us using `\n`

Escaped Characters Respected

Remember that escaped characters are still respected
These triple quotes are not an alternative to raw strings

```
str1 = """Escaped characters are still...\nrespected"""
print(str1)
```

```
Escaped characters are still...
respected
```

Injecting Variable Values into Strings

f-strings: Template Literals in Python

If you begin a string with the letter `f or F` outside of the quotation marks, you can include placeholders in your strings `{}`
Put expressions or variables inside the placeholders, and they'll be evaluated then inserted

```
unit_of_time = "seconds"
str1 = f"The {'number'} of {unit_of_time} in a day is {24 *
print(str1)  # The number of seconds in a day is 86400
```

< % Operator: Inject Variable Values w/ Defined Types into Strings >

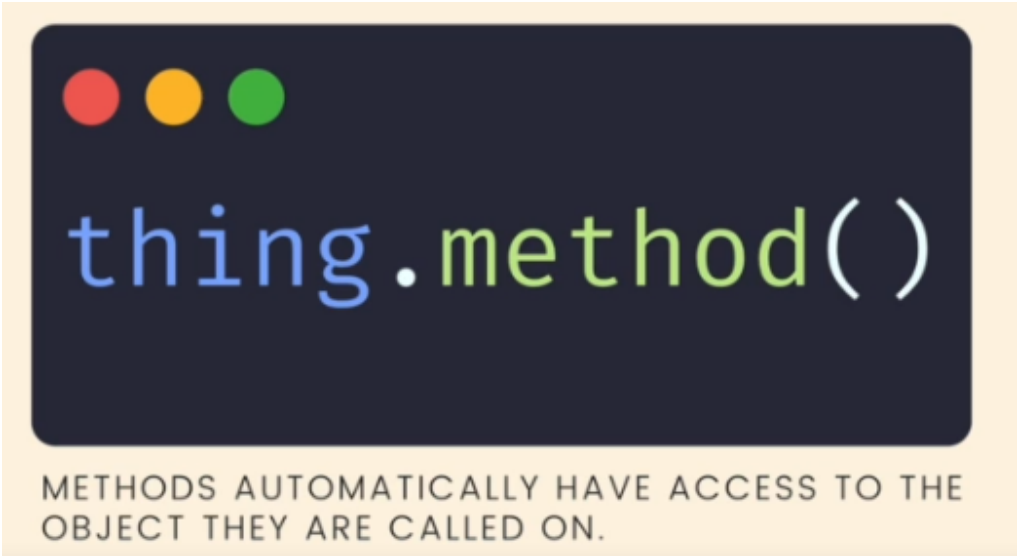
< format() Method >

https://www.w3schools.com/python/python_strings_format.asp

Methods: Make New Edited Strings

Intro to String Methods

Methods are just functions that "live on objects" - they are part of and available on all objects



Syntax is Consistent

All string methods follow the same syntax

```
"string or variable".methodName()
```

List of all String Methods

```
str.capitalize()        str.rfind()
str.endswith()          str.rindex()
str.find()              str.rjust()
str.join()              str.split()
str.lower()             str.startswith()
str.lstrip()            str.strip()
str.removeprefix()      str.swapcase()
str.removesuffix()      str.title()
str.replace()           str.upper()
```

Capitalization Methods

upper()

Returns a string where all letters are converted to upper case

```
str1 = "is THIS the full extent of your power?"    # orig s
newStr = str1.upper() # IS THIS THE FULL EXTENT OF YOUR POW
```

lower()

Returns a string where all letters are converted to upper case

```
str1 = "is THIS the full extent of your power?"    # orig s
newStr = str1.lower() # is this the full extent of your pow
```

capitalize(): Uppercase first letter of string

Returns a string where first character is upper case, and the rest are lower case

```
str1 = "is THIS the full extent of your power?"    # orig s
newStr = str1.capitalize() # Is this the full extent of you
```

title(): Uppercase first letter of each word

- Capitalizes the first letter of every word in a string
- Decapitalizes other uppercase letters in each word
- If a non-alphabetical character is encountered, the next alphabetical letter gets capitalized

```
a = "behold my POWER!" # unchanged after title() is used on
b = a.title() # Behold My Power!
c = "Door 1a opened" # unchanged after title() is used on i
d = c.title() # Door 1A Opened
```

swapcase(): Inverse Capitalization

Inverses the case of each character in a string

```
a = "LOUD VOICES whispering softly"
b = a.swapcase() # loud voices WHISPERING SOFTLY
```

Padding Methods

RESTRICTION FOR BOTH METHODS:
The padding string (parameter 2) must be exactly 1 character long

rjust() : Pad string on left

Make a string 5 characters long, and pad with the string "0" if needed

```
a = "txt" # unaffected by method
b = a.rjust(5, "0") # 00txt (adds padding to left)
```

ljust() : Pad string on right

Make a string 5 characters long, and pad with the string "0" if needed

```
a = "txt" # unaffected by method
b = a.ljust(5, "0") # txt00
```

Remove Leading & Trailing Chars in Python 3.9+

removeprefix: Remove Leading Chars

```
text = "_World"
new_text = text.removeprefix("_")
print(text) # _World
print(new_text) # World
```

removesuffix: Remove Trailing Chars

```
text = "jaystar"
new_text = text.removesuffix("star")
print(text) # jaystar
print(new_text) # jay
```

Strip Methods: Remove Leading & Trailing Chars Part 2

Strip Methods Overview

Strip methods remove leading and trailing characters

strip() demo

Remove leading and trailing whitespace, tabs, and line breaks

```
str1 = "\n Random sequence    " # original unaffected by met
newStr = str1.strip()  # "Random sequence"
```

Remove leading and trailing _ symbols

```
str1 = "__Element_115__"
newStr = str1.strip("_") # Element_115
```

Remove leading and trailing ? and $ symbols

```
str1 = "$#$#Cash $ and Status#$#$"  # original unaffected b
newStr = str1.strip("$#")  # Cash $ and Status
```

lstrip() demo

Remove leading whitespace, tabs, and line breaks

```
str1 = "\n Random sequence    " # original unaffected by met
newStr = str1.lstrip()  # "Random sequence    "
```

Remove leading _ symbols

```
str1 = "__Element_115__" # original unaffected by method
newStr = str1.lstrip("_") # Element_115__
```

Remove leading ? and $ symbols

- Notice how $# are specified in 1 string together
- This tells the method to strip out any $ or # symbols
  Not telling it to look for "#$" combinations

```
str1 = "$#$#Cash $ and Status#$#$"  # original unaffected b
newStr = str1.lstrip("$#")  # Cash $ and Status#$#$
```

rstrip() demo

Remove trailing whitespace, tabs, and line breaks

```
str1 = "\n Random sequence    " # original unaffected by met
newStr = str1.rstrip()  # "\n Random sequence" (\n is a lin
```

Remove trailing _ symbols

```
str1 = "__Element_115__" # original unaffected by method
newStr = str1.rstrip("_") # __Element_115
```

Remove trailing ? and $ symbols

```
str1 = "$#$#Cash $ and Status#$#$"  # original unaffected b
newStr = str1.rstrip("$#")  # $#$#Cash $ and Status
```

Weird Behavior

```
s = "document_type"
print(s.rstrip("_lowercase")) # document_typ
```

If using python 3.9 or later, use

```
s = "document_type"
s.removesuffix("_lowercase") # document_type
```

replace() : Replace substrings

Replaces instances of substrings inside of a string
Can replace it with a new substring or nothing at all

```
str1.replace(old, new, [count]) # old and new are required.
```

Replace all Instances of a Substring

```
str1 = "I'm going to run 3 kilometers then walk 7 kilometer
str2 = str1.replace("kilometers", "km")
# I'm going to run 3 km then walk 7 km. Overall, that's 10
```

If we provide no 3rd argument, all instances of the substring specified in argument 1 will be replaced

Replace some Instances of a Substring

If you provide a third argument, you can specify how many occurrences of the old value you want to replace

Replace first 2 instances of "kilometers"

```
str1 = "I'm going to run 3 kilometers then walk 7 kilometer
str2 = str1.replace("kilometers", "km", 2)
# I'm going to run 3 km then walk 7 km. Overall, that's 10
```

Use Case: Remove Instances

To remove all instances of a substring, make argument 2 an empty string

```
str1 = "Bills: $60, $40, $50"
str2 = str1.replace('$', "") # Bills: 60, 40, 50
```

Method is Case Sensitive

Upper/lower casing and spacing will affect whether we get matches for substrings to replace

```
str1 = "I like Apples a lot"
str2 = str1.replace('apples', "bananas") # orig string not
print(str2) # I like Apples a lot
```

join() : Turn tuples to Concatenated Strings

Join all items in a tuple using a space as a separator

```
myTuple = ("John", "Peter", "Vicky")
newString = " ".join(myTuple) #John Peter Vicky
```

Join all items in a tuple using a - as a separator (space, dash, space)

```
myTuple = ("John", "Peter", "Vicky")
x = "-".join(myTuple) #John-Peter-Vicky
```

removeprefix() : Remove Chars from String Start

Removes the substring parameter you specify from the start of the string the method's used on
If the string does not begin with that substring, it's returned as it was originally

```
new_string1 = 'mystery'.removeprefix("my") # stery
new_string2 = 'mystery'.removeprefix("xx") # mystery
```

removesuffix() : Remove Chars fr/ String End

Removes the substring parameter you specify from the end of the string the method's used on
If the string does not end with that substring, it's returned as it was originally

```
new_string1 = 'incorporated'.removesuffix("ted") # incorpor
new_string2 = 'incorporated'.removesuffix("xx") # incorpora
```

Sort String Alphabetically

Alphabetical order

```
str = "number"
ordered_letters = ''.join(sorted(str)) # bemnru
```

Reverse Alphabetical Order

```
str = "number"
ordered_letters = ''.join(sorted(str, reverse=True)) # urnm
```

Methods: Substring Occurrences

SUBSTRING OCCURRENCE:
When a smaller string appears in a larger string

```
txt = "cocoa moo" # There are 2 substring occurences of "co
```

count() : Return # of Substring Occurrences

Simply returns the number of times a substring appears
Can feed it more arguments to choose an index range to look for matches (SBE)

Demo

Count number of occurrences

```
str1 = "I love apples, apple are my favorite fruit"
x = str1.count("apple") # 2
y = str1.count("cherry") # 0
```

Count number of occurrences in a specific index range

```
str1 = "avadacadabra"
#       0123456789
a = str1.count("ad", 0, 7) # 1 (caused SBE index range)
b = str1.count("ad") # 2
```

find(): Return First Index of Substring

Overview

Inspects a string then returns the index where a substring appears
Returns -1 if the substring cannot be found

Demo

Return the index of the subring "_"

```
str1 = "__bars__"
a = str1.find('_') # 0
```

Return the index of the subring "##", but start at index 1

```
str1 = "##bars##"
a = str1.find('##', 2 ) # 6
```

Return the index of substring "ad", but start at index 3 and end at index 6

```
str1 = "avadacadabra"
#       0123456789
a = str1.find('ad', 2 ) # 2
b = str1.find('ad', 5, 7 ) # -1 (second "ad" starts at inde
c = str1.find('ad', 5, 8 ) # 6
```

rfind() : Return Last Index of Substring

Overview

Returns the index of the last occurrence of a substring - and returns -1 if no matches are found
(Same as find method, but looks for matches starting at the end of a string)

Examples

Find last occurrence of ad

```
str1 = "avadacadabra"
#       0123456789
a = str1.rfind("ad") # 6
```

Find last occurrence of ad, but start search at index 0 and end at index 7 or 8

```
str1 = "avadacadabra"
#       0123456789
a = str1.rfind("ad", 0, 7) # 2 (caused SBE index range)
b = str1.rfind("ad", 0, 8) # 6
```

Find last occurrence of "abc"

```
str1 = "akami"
a = str1.rfind("od") # -1 (no occurrences at all)
```

index() : Return First Index of Substring

Overview

- Inspects a string then returns the index where a substring appears
- If the substring cannot be found, index() throws an error

index()

This method is the exact same as find(), but it throws an error if no matches are found

Return the index of the subring "_"

```
str1 = "__bars__"
a = str1.index('_') # 0
```

Return the index of the subring "##", but start at index 1

```
str1 = "##bars##"
a = str1.index('##', 2 ) # 6
print(a)
```

Return the index of substring "ad", but start at index 3 and end at index 6

```
str1 = "avadacadabra"
#       0123456789
a = str1.index('ad', 2 ) # 2
b = str1.index('ad', 5, 8 ) # 6
c = str1.index('ad', 5, 7 ) # ERROR thown. code execution s
# SBE index ranges led to there being no match
# second "ad" starts at index 5 and ends at index 7
print("Finito") # this line is not reached due to the throw
```

rindex() : Find Last Index of Substring

Overview

- Returns the last index where a substring appears in the string this method is applied onto
- Throws an error if the substring cannot be found

Demo

Return last index of substring "at"

```
a = "Battery attack!"
b = a.rindex("at") # 8
```

Return last index of "atlas"

```
a = "Battery attack!"
b = a.rindex("atlas") # ValueError: substring not found
```

Difference b/t index & find Methods

If the substring cannot be found...

- find and rfind methods return-1
- index and rindex methods throw an error

Other Methods

startsWith() : Returns Boolean

Returns a Boolean telling if a certain string starts with a substring you provide

Parameters

Demo

```
a = "buffalo" # unchanged after method
b = a.endswith("lo") # True
c = a.endswith("xa") # False
```

endsWith() : Returns Boolean

Returns a Boolean telling if a certain string ends with a substring you provide

Parameters

Demo

```
a = "buffalo" # unchanged after method
b = a.endswith("lo") # True
c = a.endswith("x") # False
```

check for substring (in operator)

```
string = "Hello, world!"
substring = "world"
is_substring = substring in string
print(is_substring) # True
```

split() : Turn String into List

Formula

```
string.split(separator, maxsplit)
```

Parameters

Demo

Split the string using whitespace as a separator

```
txt = "hi world,  how are things?"
x = txt.split() # ['hi', 'world,', 'how', 'are', 'things?']
```

Split the string, using a comma followed by a space as a separator:

```
txt = "hello, my name is Peter, I am 26 years old"
x = txt.split(", ") #  ['hello', 'my name is Peter', 'I am
```

Limiting Splits + rsplit()

If you want to cap the number of splits, use the maxsplit parameter

- You can start the first split on the left and continue towards the right using split(), method
- Alternatively, you can start splits on the right and continue towards the left with `rsplit()`

```
txt = "What's new Scooby Doo?"
# Take 2 splits starting from the left, using a whitespace
x = txt.split(" ", 2) # ["What's", 'new', 'Scooby Doo?']
# Take 2 splits starting from the right, using a whitespace
y = txt.rsplit(" ", 2) # ["What's new", 'Scooby', 'Doo?']
```

LISTS

Lists: Python Arrays

Intro to Lists

What are Lists?

Lists are a data structure that holds index-ordered data - just like JS arrays

- They can hold any data type, or multiple different ones
- Duplicate items are allowed
- You can change the list items - add, remove, or reorder .. etc

Declaring Lists

Standard Syntax

```
list1 = ["apple", "cherry", "orange"];
print(list1) # ['apple', 'cherry', 'orange']
```

List Constructor Syntax

```
list1 = list(("apple", "cherry", "orange")) # doubble brack
print(list1) # ['apple', 'cherry', 'orange']
```

Declare empty list

```
empty_list_1 = []
empty_list_2 = list()
```

len() : Get List Length

Use then `len()` function to get a list length

```
list1 = ["apple", "cherry", "orange"];
print(len(list1)) # 3
```

Access List Items

REMEMBER

1. Lists are indexed starting at 0
2. Ranges are outer bound exclusive (up to but not including second index)
3. Using bracket notation out of bounds of a list's index range may throw an error

Access one List Item

Positive index:
0 is the first item, 1 is the second item

Negative indexes:
-1 is the last item, -2 is the second last item

```
list1 = ["python", "javascript", "rust", "go"]
a = list1[2] # rust
b = list1[-1] # go
c = list1[-3] # javascript
```

Access Positive List Range

Access List items at index 1 to 3

```
list1 = ["python", "javascript", "rust", "go", "ruby"]
subarray = list1[1:4] # outer bound exclusive (index 3 is t
print(subarray) # ['javascript', 'rust', 'go']
```

Access items from index 3 and beyond

```
list1 = ["python", "javascript", "rust", "go", "ruby"]
subarray = list1[3:] # unspecified end bound (assumes you w
print(subarray) # ['go', 'ruby']
```

Access Negative List Range

Access list items from 5th last to and including 3rd last

```
list1 = ["python", "javascript", "rust", "go", "ruby"]
subarray = list1[-5:-2] # must use -2 to include the third
print(subarray) # ['python', 'javascript', 'rust']
```

Access items from 4th last to the very last

```
list1 = ["python", "javascript", "rust", "go", "ruby"]
subarray = list1[-4:]
print(subarray) # ['javascript', 'rust', 'go', 'ruby']
```

Mixed List Ranges +-

Go from 4th last to and including index 3

```
list1 = ["python", "javascript", "rust", "go", "ruby"]
subarray = list1[-4:4]
print(subarray) # ['javascript', 'rust', 'go']
```

Safest way to Access List Item

Trying to access a list item that's out of bounds causes an error
To avoid this, use ternary

Unsafe

```
my_list = [0, 1]
a = my_list[5]  #! IndexError: list index out of range
```

Safe

```
my_list = [0, 1]
a = my_list[5] if 5 < len(my_list) else None  # None
```

Complex Example: Access Nested List Items Safely

```
payload = {
    "results": {
        "data": {
            "name": "Pterano",
            "age": 60,
            "stats": [0, 1, 2, 3]
        }
    }
}
# Access deeply nested dict properties, and set equal to No
name = payload.get("results", {}).get("data", {}).get("name
species = payload.get("results", {}).get("data", {}).get("s
# Access deeply nested list properties, and set equal to No
stats = payload.get("results", {}).get("data", {}).get("sta
ninth_stat = stats[8] if 8 < len(stats) else None
print(name)  # Pterano
print(species)  # None
print(ninth_stat)  # None
```

Check if Value is or Isn't in a List

SIMPLE VALUE CHECKS ONLY:
For more complicated operations like checking if an object with a certain property exists, use a custom function or a library like Pandas

in operator: Check if value is inside List

Returns True if an item is inside the list - returns False otherwise

```
nums = [99, 4, 52, 61]
booly1 = 99 in nums    # True because 99 is inside
booly2 = 5000 in nums # False because 5000 is not inside
```

not in operator: Check if value is not Inside List

Returns True if an item is not inside the list - returns False otherwise

```
nums = [99, 4, 52, 61]
booly1 = 99 not in nums    # False because it is in the list
booly2 = 5000 not in nums # True because it is not in the l
```

Practical Use Case: Conditionals

The in operator can be used to check if certain values exist in an array

```
list1 = ["python", "javascript", "rust", "go", "ruby"]
condition = "swift" in list1
print(condition) # False
# Practical use case: conditionals
if "swift" in list1:
    print("I should learn swift")
```

Overwrite List Item Values

Overwrite 1 List Item

Overwrite the item at index 2

```
langs = ["Python", "javascript", "Rust", "Go", "Ruby"]
langs[1] = "SQL"
print(langs) # ['Python', 'SQL', 'Rust', 'Go', 'Ruby']
```

Overwrite the last index (using negative indexes)

```
langs = ["Python", "Javascript", "Rust", "Go", "Ruby"]
langs[-1] = "SQL"
print(langs) # ['Python', 'Javascript', 'Rust', 'Go', 'SQL'
```

Restriction: Must use Indexes that already Exist in the List

RESTRICTION:
You cannot overwrite a list item using an index that does not exist yet - it produces an error

```
langs = ["Python", "Javascript", "Rust", "Go", "Ruby"]
langs[22] = "SQL" # IndexError: list assignment index out o
print(langs)
```

Overwrite Range of Values

Change the list values at indexes 2 and 3

```
langs = ["Python", "Javascript", "Rust", "Go", "Ruby"]
langs[2:4] = ["Coffeescript", "Cobol"] # outerBound - inner
print(langs) # ['Python', 'Javascript', 'Coffeescript', 'Co
```

You can replace a range of items with less of items if you want
Ex. Replace index 3 and beyond with "Erlang"

```
langs = ["Python", "Javascript", "Rust", "Go", "Ruby"]
langs[2:] = ["Erlang"]
print(langs)  # ['Python', 'Javascript', 'Erlang']
```

You can replace a range of items with more items if you want
Ex. Replace index 3 and beyond with "Erlang"

```
foods = ["Pie", "Oatmeal", "Gumbo", "Soup"]
# Replace items at index 1 and 2 (Oatmeal and Gumbo) with f
foods[1:3] = ["Pizza", "Sushi", "Steak", "Tacos"]
print(foods)  # ['Pie', 'Pizza', 'Sushi', 'Steak', 'Tacos',
```

You can replace a range of items with nothing

```
thislist = ["blackberry", "apple", "android", "motorola"]
thislist[1:3] = [] # replaces items at index 1 and 2 (outer
print(thislist) # ['blackberry', 'motorola']
```

Niche Behavior when Overwriting Ranges

If you try to replace entries using a range beyond the list's current index range, it just adds them to the end

```
langs = ["Python", "Javascript", "Rust", "Go", "Ruby"]
langs[80:100] = ["Coffeescript", "Cobol"]
print(langs)  # ['Python', 'Javascript', 'Rust', 'Go', 'Rub
```

Looping through Lists

M1: Standard for Loop

Use if all you care about is accessing the values in the list

```
phones = ["blackberry", "apple", "android", "motorola"]
for brand in phones:
  print(brand)
```

```
apple
android
motorola
```

M2: Access indexes during For Loop

Use if you want to access the values and indexes of each list item

```
phones = ["blackberry", "apple", "android", "motorola"]
for i in range(len(phones)):
  print([i, phones[i]])
```

```
[0, 'blackberry']
[1, 'apple']
[2, 'android']
[3, 'motorola']
```

M3: While Loop

Use if you want to access the values and indexes of each list item
Or if you need more control over the number of iterations

```
phones = ["blackberry", "apple", "android", "motorola"]
i = 0
while i < len(phones):
    print([i, phones[i]])
    i += 1
```

```
[0, 'blackberry']
[1, 'apple']
[2, 'android']
[3, 'motorola']
```

List Comprehension: Create new List based off another

PURPOSE:
The return value is a new list, leaving the old list unchanged

Syntax & Variants

No conditional syntax

```
newlist = [new_value_expression for item in iterable]
```

if condition syntax

```
newlist = [new_value_expression for item in iterable if con
```

if else condition syntax

```
newlist = [new_value_expression if condition else fallback_
```

Demo

Make a new list with capitalized items
Apply to every list item, no matter what (no conditions)

```
phones = ["blackberry", "apple", "android", "motorola"]
# without conditional
new_phones = [brand.upper() for brand in phones]
print(new_phones) # ['BLACKBERRY', 'APPLE', 'ANDROID', 'MOT
```

Make a new list with capitalized items, but only for names that start with the letter "a"

```
phones = ["blackberry", "apple", "android", "motorola"]
new_phones = [brand.upper() for brand in phones if brand.st
print(new_phones)  # ['APPLE', 'ANDROID']
```

Fallback Expression

You can use `else` to leave expressions that will be used instead of `new_value_expression` for when the if condition fails
The syntax changes a bit

```
phones = ["blackberry", "apple", "android", "motorola"]
new_phones = [brand.upper() if brand.startswith("a") else "
print(new_phones) # ['Irrelevant', 'APPLE', 'ANDROID', 'Irr
```

Longhand Alternative

You could always just loop over a list normally and build a new one

GOAL:
Make a new list with capitalized items, but only for names that start with the letter "a"

Shorthand with List Association (1 line)

```
phones = ["blackberry", "apple", "android", "motorola"]
# without conditional
new_phones = [brand.upper() for brand in phones if brand.st
print(new_phones)  # ['APPLE', 'ANDROID']
```

Longhand (4 lines)

```
phones = ["blackberry", "apple", "android", "motorola"]
# without conditional
new_phones = []
for brand in phones:
    if brand.startswith("a"):
        new_phones.append(brand.upper())
print(new_phones)
```

Slice an Array

The indexes you supply determine what values your new sliced list will have
The end index is not included

```
numbers = [0, 1, 2, 3, 4, 5]
numbers[1:4]    # [1, 2, 3]      # From index 1 to 3
numbers[:3]     # [0, 1, 2]       # Start to index 2
numbers[0:3]     # [0, 1, 2]       # Start to index 2
numbers[2:]     # [2, 3, 4, 5]    # Index 2 to end
numbers[::2]    # [0, 2, 4]       # Every second element
numbers[::-1]   # [5, 4, 3, 2, 1, 0]  # Reverse
# Negative indices count from end
numbers[-3:]    # [3, 4, 5]       # Last three elements
numbers[:-2]    # [0, 1, 2, 3]    # All except last two
```

Methods that Edit Lists

insert() : Inject List Item at index i

Inject the value "Physics" into the list at index 1

```
courses = ["Math", "Science", "Religion", "English"]
courses.insert(1, "Physics")
print(courses) # ['Math', 'Physics', 'Science', 'Religion',
```

Math	Science	Religion	English		
0	1	2	3		

Math	Physics	Science	Religion	English	
0	1	2	3		

append() : Add List value to end

```
courses = ["Math", "Science", "Religion", "English"]
courses.append("Physics")
print(courses) # ['Math', 'Science', 'Religion', 'English',
```

extend() : Concatenate Lists

Combine Lists

```
regula = ["apple", "banana"]
tropical = ["pineapple", "papaya"]
regula.extend(tropical)
print(regula) # ['apple', 'banana', 'pineapple', 'papaya']
```

Combine List with other Iterables

Feel free to combine lists with other iterables

List + tuple

```
thislist = ["apple", "banana", "cherry"]
thistuple = ("kiwi", "orange")
thislist.extend(thistuple)
print(thislist) # ['apple', 'banana', 'cherry', 'kiwi', 'or
```

List + string

```
thislist = ["apple", "banana", "cherry"]
stringy = "SUPER"
thislist.extend(stringy)
print(thislist) # ['apple', 'banana', 'cherry', 'S', 'U', '
```

del keyword: Destroy Entire List & Variable

The `del` keyword allows you to destroy the variable holding the list
If you try to reference it later, you'll get an error

```
thislist = ["apple", "banana", "cherry"]
del thislist
print(thislist) # NameError: name 'thislist' is not defined
```

clear() : Empty the list

The `clear()` method empties out the list, but keeps it in existence

```
thislist = ["apple", "banana", "cherry"]
thislist.clear()
print(thislist) # []
```

pop() + del: Remove List Items at index i

Method 1: pop()

Remove item at index 2

```
thislist = ["blackberry", "apple", "android", "motorola"]
thislist.pop(2) # pop() with positive index
print(thislist) # ['blackberry', 'apple', 'motorola']
```

Remove last item in list

```
thislist = ["blackberry", "apple", "android", "motorola"]
thislist.pop() # pop() with no arg
print(thislist) # ['blackberry', 'apple', 'android']
```

Remove third last item in list

```
thislist = ["blackberry", "apple", "android", "motorola"]
thislist.pop(-3) # pop() with negative index
print(thislist) # ['blackberry', 'android', 'motorola']
```

Method 2: del keyword

Delete list item at index 1

```
thislist = ["apple", "banana", "cherry"]
del thislist[1]
```

```
print(thislist) # ['apple', 'cherry']
```

WARNING:
This produces an error if you try to delete an item at an index out of range on the current list

```
thislist = ["apple", "banana", "cherry"]
del thislist[5] # IndexError: list assignment index out of
```

Remove Multiple Items via Ranges

Use ranges to remove multiple items in a list

```
thislist = ["blackberry", "apple", "android", "motorola"]
thislist[1:3] = [] # replaces items at index 1 and 2 with n
print(thislist) # ['blackberry', 'motorola']
```

Returns Removed Value

The pop method actually returns the value being removed from the list
May be useful

```
thislist = ["blackberry", "apple", "android", "motorola"]
print(thislist.pop()) # motorola
print(thislist) # ['blackberry', 'apple', 'android']
```

sort() : Sort the Original List

Numeric Order

Lowest to highest

```
nums = [44, 50, 90, 12]
nums.sort()
print(nums) # [12, 44, 50, 90]
```

Highest to lowest

```
nums = [44, 50, 90, 12]
nums.sort(reverse=True)
print(nums) # [90, 50, 44, 12]
```

Alphabetical order

Alphabetical order

```
fast = ["Arbys", "Zaxbys", "Popeyes", "Burger King"]
fast.sort()
print(fast) # ['Arbys', 'Burger King', 'Popeyes', 'Zaxbys']
```

Reverse alphabetical order

```
fast = ["Arbys", "Zaxbys", "Popeyes", "Burger King"]
fast.sort(reverse=True)
print(fast)  # ['Zaxbys', 'Popeyes', 'Burger King', 'Arbys'
```

Based on Custom Logic

Custom logic: Sort based on the length of a word

```
fast = ["KFC", "DQ", "Royal Chinese Cuisine", "Popeyes"]
fast.sort(key=lambda word: len(word)) # lambdas are cleaner
print(fast) # ['DQ', 'KFC', 'Popeyes', 'Royal Chinese Cuisi
```

Based on Nested Property

Sort the list in alphabetical order, in regards to the dictionary item's make property

```
from operator import itemgetter
cars = [
    {'car': 'Ford', 'year': 2005},
    {'car': 'Mitsubishi', 'year': 2000},
    {'car': 'BMW', 'year': 2019},
    {'car': 'Volkswagen', 'year': 2011}
]
cars.sort(key=itemgetter('car'))
print(cars)
```

```
[
    {'car': 'BMW', 'year': 2019},
    {'car': 'Ford', 'year': 2005},
    {'car': 'Mitsubishi', 'year': 2000},
    {'car': 'Volkswagen', 'year': 2011}
]
```

reverse() : Reverse a List

Mutate Original

The `reverse()` method perma-mutates a list and reverses its order, then returns None

```
phones = ["a", "b", "c"]
phones.reverse() # perma mutates the original data structur
print(phones) # ['c', 'b', 'a']
```

Create new Reversed Array

If you want to avoid permanently mutating the original

Negative Range

```
phones = ["a", "b", "c"]
new = phones[::-1]
print(new)       # ['c', 'b', 'a']
print(phones)    # ["a", "b", "c"]
```

Reversed Method

```
a = [1, 5, 6]
b = list(reversed(a))
print(a) # [1, 5, 6]
print(b) # [6, 5, 1]
```

remove() : Remove first matching Element

Remove the first instance of a value in a list

```
nums = [44, 50, 90, 12]
nums.remove(90)
print(nums) # [44, 50, 12]
```

ERROR WARNING
Trying to remove a value not in the list will cause an error

```
nums = [44, 50, 90, 12]
nums.remove(200) # ValueError: list.remove(x): x not in lis
```

Methods that Generate New Lists based off Existing Ones

copy() : Shallow clone

Create a shallow clone of a list using the copy() method
Be aware that edits to a nested reference object will affect both the clone and the original

```
phones = ["blackberry", "apple", "android", [44, 81]]
shallow_clone = phones.copy()
print(phones)
print(shallow_clone)
```

```
['blackberry', 'apple', 'android', [44, 81]]
['blackberry', 'apple', 'android', [44, 81]]
```

Show how the cloned list is shallow

```
phones = ["blackberry", "apple", "android", [44, 81]]
shallow_clone = phones.copy()
shallow_clone[3][0] = 50 # affects both
print(phones)
print(shallow_clone)
```

```
['blackberry', 'apple', 'android', [50, 81]]
['blackberry', 'apple', 'android', [50, 81]]
```

sorted() : Generate New Sorted List

This method creates a new list without affecting the original

Standard

```
nums = [9, 1, 8]
sortedNums = sorted(nums)
print(nums) # [9, 1, 8]
print(sortedNums) # [1, 8, 9]
```

Custom Sorting Logic

Sort by name length from longest → shortest

```
names = ["BritneyBritney", "AJ", "Timmy"]
sorted_names = sorted(names, key=lambda name: len(name), re
print(sorted_names)  # ['BritneyBritney', 'Timmy', 'AJ']
```

map() : Just like JS

Generate a new list by applying actions onto the items of an already existing list
This method leaves the original list unmutated

Simple Example

With simple logic, it's cleaner to use lambdas over regular methods

EXAMPLE:
Generate a new list whose values are double that of the original

```
nums = [8, 1, 30, 12]
double_nums = list(map(lambda num: num*2, nums))
print(double_nums) # [16, 2, 60, 24]
print(nums) # [8, 1, 30, 12]
```

Complex Example

Double the odd numbers while tripling the even numbers in a list
This could be done using a lambda, but for more advanced logic you're better off defining a function and passing it as an argument

```
nums = [4, 1, 18, 7]
def double_even_only(num):
    if num % 2 == 0:
        return num * 2
    else:
        return num
edited_nums = list(map(double_even_only, nums))
print(edited_nums)  # [8, 1, 36, 7]
print(nums)  # [4, 1, 18, 7]
```

filter()

Filter Callback Requirements

Define a method or lambda that returns a boolean

- True means the value passes through the filter

- False means the value gets filtered out and excluded

Simple Example

With simple logic, it's cleaner to use lambdas over regular methods

EXAMPLE:
Remove all odd numbers from a list

```
nums = [4, 1, 18, 7, 5]
even_only = list(filter(lambda num: num % 2 == 0, nums))
print(even_only) # [4, 18]
print(nums) # [4, 1, 18, 7, 5] The original list is left in
```

Complex Example

When using complex logic, it's better to define a function instead of using a lambda

EXAMPLE:
Remove all names that contain either of these substrings: `["ud", "ka"]`

```
names = ["anna", "karen", "mahmud", "kimmy"]
def exclude_certain_names(name):
    if "ud" in name:
        return False
    elif "ka" in name:
        return False
```

```
    else:
        return True
filtered_names = list(filter(exclude_certain_names, names))
print(filtered_names)  # ['anna', 'kimmy']
print(names)  # ['anna', 'karen', 'mahmud', 'kimmy']    The
```

Other List Methods

min() + max() : Get Max & Min Values in List

```
nums = [9, 1, 8]
minimum = min(nums) # 1
maximum = max(nums) # 9
```

index() : Return index of certain value in list

Find the index of "apple" in the list

```
phones = ["blackberry", "apple", "android", "motorola"]
location = phones.index("apple") # 1
```

ERROR WARNING:
Looking for a value that is not present in the list will cause an error to be thrown

```
phones = ["blackberry", "apple", "android", "motorola"]
location = phones.index("huawei") # ValueError: 'huawei' is
```

count() : Number of Times a Value Appears in a List

Count number of times "apple" appears in the list

```
phones = ["apple", "apple", "huawei", "android", "apple"]
num = phones.count("apple") # 3
```

INTERESTING PERK:
Method works with any data type - even complex ones like dictionaries or

other lists

```
phones = ["blackberry", {'a': 44, 'b': 81}, "android", {'a'
num_occurrences = phones.count({'a': 44, 'b': 81})
print(num_occurrences)  # 2
```

= = = = = = = DICTIONARIES = = = = = = =

Dictionaries: Python Objects

Intro to Dictionaries

Python dictionaries are a data structure that have key value pairs
Each non-variable key must be defined with quotation marks - unlike in JS

Declare Dictionaries

Standard Syntax

```
keyboard = {
    "brand": "Logitech",
    "model": "K860",
    "color": "Black",
}
```

Constructor Syntax

```
keyboard = dict(brand="Logitech", model="K860", color="Blac
```

Dynamic keys and Values
You can assign properties and associated values using variables

```
key = "color"
val = "black"
keyboard = { key: val }
print(keyboard) # {'color': 'black'}
```

Properties of dictionaries

- No duplicate properties allowed
  If you try to assign a value to a property that's already present, you'll just overwrite the value

- Key value pairs are ordered as of Python 3.7
  See OrderedDict lesson for more about that

Get List of Keys, Values, or KVP's

Number of Key Value Pairs

The len function works on dictionaries too

```
keyboard = {
    "brand": "Logitech",
    "model": "K860",
    "color": "Black",
}
print(len(keyboard)) # 3
```

keys() : List of Keys

Similar to Object.keys() in JS

```
my_dict = {'a': 1, 'b': 2, 'c': 3}
keys_list = list(my_dict.keys())
print(keys_list) # ['a', 'b', 'c']
```

values() : List of Values

Similar to Object.values() in JS

```
my_dict = {'a': 1, 'b': 2, 'c': 3}
values_list = list(my_dict.values())
print(values_list) # [1, 2, 3]
```

Alternatively, you can create a set of values to remove duplicate values from appearing

```
keyboard = {
    "brand": "Logitech",
    "model": "Black",
    "color": "Black", # duplicate value
}
keys = set((keyboard.values())) # must use list constructor
print(keys) # {'Black', 'Logitech'}
```

items() : List of Tuples Containing KVPs

Similar to Object.entries() in JS

```
my_dict = {'a': 1, 'b': 2, 'c': 3}
keys_list = list(my_dict.items())
print(keys_list) # [('a', 1), ('b', 2), ('c', 3)]
```

You can extract the key and value from each tuple using unpacking

```
students = {'Mallory': 21, 'Ben': 17}
kvps = list(students.items()) # [('Mallory', 21), ('Ben', 1
for key, value in kvps:
    print(f"{key} is {value} years old")
# Mallory is 21 years old
# Ben is 17 years old
```

Check if key is or is not Present

in Operator: Return True if Property is Present

```
person = { "age": 27, "height": "5 ft 9"}
booly1 = "age" in person  # True
booly2 = "gender" in person  # False
```

not in Operator: Return True if Property is Absent

```
person = { "age": 27, "height": "5 ft 9"}
booly1 = "age" not in person  # False
booly2 = "gender" not in person  # True
```

Use Case: Conditionals

```
person = { "age": 27, "height": "5 ft 9" }
if "gender" not in person:
    print("Need to specify your gender")
```

Access Dictionary Values that 100% are Present

ERROR PRONE: Access KVPs that 100% are Present

Bracket notation (throws error if KVP does not exist)

```
person = { "name": "Jason" }
name = person["name"] # Jason
```

WARNING:
Bracket Notation can throw Errors if a KVP is not present
Maybe that's what you want, and maybe it isn't (use accordingly)

```
person = {}  # empty dictionary
```

```
middle_name = person["middle_name"] # KeyError: 'middle_nam
```

Access Dictionary Values that May not be Present

In cases where a dictionary may or may not have a certain key value pair, use the get() method

- It returns `None` if the KVP is not present
- You can chain multiple get calls together with default values in case you're trying to access deeply nested properties that may not exist (last example in this lesson)

Access KVPs that May not be Present

Get method (best choice)

```
person = {} # empty dictionary
name = person.get("middle_name") # None
```

Alternative: Bracket Notation + ternary
(not practical for deeply nested property values)

```
person = {}  # empty dictionary
middle_name = person["middle_name"] if "middle_name" in per
print(middle_name) # None
```

Complex Example: Access Deeply Nested Dict Properties Safely

```
payload = {
    "results": {
        "data": {
            "name": "Pterano",
            "age": 60,
            "stats": [0, 1, 2, 3]
        }
    }
}
# Access deeply nested dict properties, and set equal to No
name = payload.get("results", {}).get("data", {}).get("name
species = payload.get("results", {}).get("data", {}).get("s
# Access deeply nested list properties, and set equal to No
stats = payload.get("results", {}).get("data", {}).get("sta
ninth_stat = stats[8] if 8 < len(stats) else None
print(name)  # Pterano
print(species)  # None
print(ninth_stat)  # None
```

Fallback Syntax

You can provide the .get method a second argument, and have that be the fallback value returned if the KVP does not exist
Is a useful replacement for the `or` operator

```
freq = {}
r = freq.get(1, "Fallback")
print(r) # Fallback
```

Set or Overwrite KVP

You can set or overwrite property values using 2 methods
Dot notation is not an option

Bracket Notation

```
person = {
    "name": "Jason",
    "age": 27,
    "job": "Software Engineer",
}
person["age"] = 99 # overwrite existing kvp
person["last_name"] = "XB" # define new kvp
print(person) # {'name': 'Jason', 'age': 99, 'job': 'Softwa
```

Update Method

```
person = {
    "name": "Jason",
    "age": 27,
    "job": "Software Engineer",
}
person.update({"age": 99}) # overwrite existing kvp
person.update({"last_name": "XB"}) # define new kvp
print(person) # {'name': 'Jason', 'age': 99, 'job': 'Softwa
```

Remove Specific KVP

del keyword

Remove a KVP that may or may not be present

```
person = {
    "name": "Jason",
    "age": 27,
    "job": "Software Engineer",
}
person.pop("age") if "age" in person else None # use ternar
print(person)  # {'name': 'Jason', 'job': 'Software Enginee
```

pop() Method

```
person = {
    "name": "Jason",
    "age": 27,
    "job": "Software Engineer",
}
person.pop("age") if "age" in person else None # use ternar
print(person)  # {'name': 'Jason', 'job': 'Software Enginee
```

Warning: Use Ternaries for Protection vs Non Existent KVP's

The del keyword and pop methods both throw errors if you try to remove a KVP that is not present

```
person = {
    "name": "Jason",
```

```
    "age": 27,
    "job": "Software Engineer",
}
del person["height"] # KeyError: 'height'
```

popItem() : Remove last inserted KVP

popItem removes the most recently inserted key value pair in a dictionary

Demo

Example 1:

```
person = { "age": 27, "height": "5 ft 9"}
person["name"] = "Jason"
person.update({"last_name": "XB"})
print(person)
person.popitem() # removes most recently injected item (las
print(person)
```

```
{'age': 27, 'height': '5 ft 9', 'name': 'Jason', 'last_name': 'XB'}
{'age': 27, 'height': '5 ft 9', 'name': 'Jason'}
```

Example 2:
If you have only defined an object and have not injected any new KVP's the one defined last is removed

```
person = { "age": 27, "height": "5 ft 9"}
print(person)
person.popitem() # removes most recently injected item (las
print(person)
```

```
{'age': 27, 'height': '5 ft 9'}
{'age': 27}
```

Useful Return

The popitem method returns a tuple containing the key and value of the removed KVP

```
person = { "age": 27, "height": "5 ft 9"}
person["name"] = "Jason"
print(person.popitem()) # returns a tuple
```

```
('name', 'Jason')
```

Only useful on Python 3.7 and Onwards

In earlier versions of Python, this would remove a random KVP from the dictionary
This is due to the fact that dictionaries only became ordered on 3.7 and future versions

clear() : Remove all KVP's

Removes all key value pairs and produces an empty object

```
person = { "age": 27, "height": "5 ft 9"}
person.clear()
print(person) # {}
```

copy() : Shallow Clone Object

The method `copy()` shallow clones a dictionary

```
person1 = {
    "name": "Ben Shaw",
    "age": 16,
    "profile_data": {"employee_num": 10004, "position": "As
}
person2 = person1.copy() # shallow clone
print("SHALLOW CLONE SHOWCASE")
person1["name"] = "Alex"
print(person1)
print(person2)
print()
print("PROOF THAT THIS IS NOT A DEEP CLONE")
person1["profile_data"]["position"] = "Burger Flipper"
print(person1)
print(person2) # position would not be "Burger Flipper" if
```

[image]

Dictionary Methods + Adv Tech

Spread Operator

```
need = {"apple": 2, "milk": 8}
want = {"ps5": 800}
all = {**need, **want}
print(all)  # {'apple': 2, 'milk': 8, 'ps5': 800}
```

fromkeys() : Generate Dictionary using List of keys + Default Value

This method lets you create dictionaries rapidly with...

- A tuple of keys to use as unique properties
- A variable to use as the value for each propery

Parameters

SYNTAX

```
dict.fromkeys(keys, value)
```

PARAMS

Demo

```
keys = ("name", "age", "job")
```

```
new_dict = dict.fromkeys(keys, "unknown")
print(new_dict) # {'name': 'unknown', 'age': 'unknown', 'jo
```

Practical Use Case: Frequency Object

```
frequency_object = dict.fromkeys(('a', 'b', 'c', 'd', 'e',
```

```
{'a': None, 'b': None, 'c': None, 'd': None, 'e': None, 'f': None, 'g': None, 'h': None, 'i': None, 'j'
: None, 'k': None, 'l': None, 'm': None, 'n': None, 'o': None, 'p': None, 'q': None, 'r': None, 's': No
ne, 't': None, 'u': None, 'v': None, 'w': None, 'x': None, 'y': None, 'z': None, 'A': None, 'B': None,
'C': None, 'D': None, 'E': None, 'F': None, 'G': None, 'H': None, 'I': None, 'J': None, 'K': None, 'L':
 None, 'M': None, 'N': None, 'O': None, 'P': None, 'Q': None, 'R': None, 'S': None, 'T': None, 'U': Non
e, 'V': None, 'W': None, 'X': None, 'Y': None, 'Z': None}
```

setdefault() : Set KVP if key is Absent

Explanation

IF THE KEY EXISTS

- Method returns the associated value of that key

IF THE KEY DOES NOT EXIST

- Method returns the value for it's 2nd parameter
- Sets the key equal to the 2nd parameter value

ONLY CONSIDERS KEY PRESCENCE
Remember, this method checks if the key exists
It will not overwrite the KVP even if that key has a value set to None or any other falsy

Demo

```
car = {
  "make": "Nissan",
  "model": "Leaf",
```

```
  "year": 2016
}
x = car.setdefault("model", "Altima") # Leaf (model key is
y = car.setdefault("kilometers", 120000) # 120000 (kilomete
print(car) # {'make': 'Nissan', 'model': 'Leaf', 'year': 20
```

Counter: Special Data Structure Good for Leetcode

Declare a Counter

The Counter data structure is a special dictionary that acts exactly like a frequency object

- It has access to all the regular dict methods, along with additional ones
- It intuitively returns 0 if you try to access a KVP that doesn't exist
  This is unlike a regular dict which throws a keyerror

Big O Scores

Generate Counter based on a String

Only include letters within string

```
from collections import Counter
s1 = "jsinzz"
freq = Counter(s1) # Counter({'z': 2, 'j': 1, 's': 1, 'i':
```

Include counts for all letters in string

```
from collections import Counter
import string
```

```
s1 = "jsinzz"
freq = Counter({char: 0 for char in string.ascii_lowercase}
freq.update(s1)
print(freq) # {'z': 2, 'i': 1, 'j': 1, ... etc }
```

Declare Empty Counter

```
from collections import Counter
c1 = Counter({})
```

Declare Empty Counter with Initialized Values

Sometimes you know what the keys in your frequency object will be, and you want to prep their values ahead of time
Ex. `{ "a": [], "b": []}`

```
import string
from collections import Counter
freq = Counter({char: [] for char in string.ascii_lowercase
```

```
Counter({'a': [], 'b': [], 'c': [], 'd': [], 'e': [], 'f': [], 'g': [], 'h': [], 'i': [], 'j': [], 'k': [], 'l': [],
'm': [], 'n': [], 'o': [], 'p': [], 'q': [], 'r': [], 's': [], 't': [], 'u': [], 'v': [], 'w': [], 'x': [], 'y': [],
'z': []})
```

Get Most Common Letters

```
from collections import Counter
s1 = "jsinzz"
freq = Counter(s1) # Counter({'z': 2, 'j': 1, 's': 1, 'i':
# Return list of tuples sorted by count (highest → lowest)
common = freq.most_common() # [('z', 2), ('j', 1), ('s', 1)
highest_count_kvp = freq.most_common(1) # [('z', 2)]
```

```
highest_count_num = freq.most_common(1)[0][1]  # 2
```

Feature: Edit Values that don't Exist Yet

The best feature about a counter object is that it doesn't throw a KeyError when you try to access or edit a KVP which does not exist yet

```
import string
from collections import Counter
freq = Counter({char: 0 for char in string.ascii_lowercase}
print(freq["5"]) # 0
freq["5"] += 1 # No errors thrown!
print(freq["5"]) # 1
```

Feature: Add or Subtract 2 Counter Dicts

Rule No Negativity

The counter dict does not store negative or zero counts

- If a KVP reaches 0 or a negative, it will be removed from the counter
- This applies to both addition and subtraction

Demonstration

Subtraction

```
from collections import Counter
```

```
counter1 = Counter({'a': 3, 'b': 2, 'c': 1})
counter2 = Counter({'a': 1, 'b': 2, 'q': 1}) # subtracting
diff = counter1 - counter2
print(diff) # Counter({'a': 2, 'c': 1})
```

Addition

```
from collections import Counter
# Example counters
counter1 = Counter({'a': 3, 'b': 2, 'c': 1})
counter2 = Counter({'a': -10, 'b': 2, 'q': 1}) # adding a:
sum = counter1 + counter2
print(sum) # Counter({'b': 4, 'c': 1, 'q': 1})
# Does not mutate any of the counters used in the arithmeti
print(counter1) # Counter({'a': 3, 'b': 2, 'c': 1})
```

= = = = = = = = = SETS

= = = = = = = = =

Sets: Unordered List holding Immutable Data with no Duplicates

Sets Explained

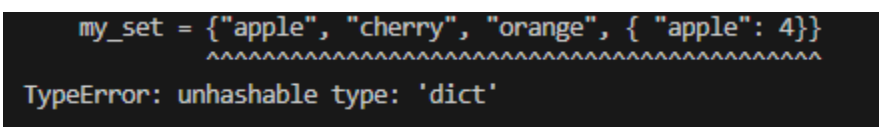
Characteristics

COMPARED TO JS SETS
Python sets are ordered while JS sets actually save the order of insertion

Holds Immutable Values only

If you try to store a mutable value inside of a set, Python throws an error
Only store immutable ones like stings, integers, floats, and tuples

```
my_set = {"apple", "cherry", "orange", { "apple": 4 }}
```



Sets are Reference Objects

This means that when you assign a set to a new variable, both variables refer to the same object in memory.
Any modifications made through one variable will be reflected when accessing the set through the other variable

```python
# Create a set
original_set = {1, 2, 3}
# Assign it to a new variable
new_set = original_set
# Modify the set using the new variable
new_set.add(4)
# Both variables reflect the change
print(original_set)  # Output: {1, 2, 3, 4}
print(new_set)       # Output: {1, 2, 3, 4}
```

Sets can Hold Instances of Classes

Class instances are hashable and can be placed inside of sets- this is useful for Leetcode problems

Demo

EXAMPLE:
Perhaps we want to check if a list node (instance of a node class) has already appeared in a linked list

```python
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None
list_node = ListNode(5)
my_set = {list_node}
is_present_1 = list_node in my_set    # True
```

```
# False, because we're checking for a new node stored in a
is_present_2 = ListNode(5) in my_set
```

Why Sets can Hold Class Instances

OVERLY SIMPLIFIED RULE OF SETS:
Sets can only hold immutable data - meaning their values cannot be changed

ACCURATE RULE OF SETS
Sets can only hold hashable objects - and hashable objects are usually immutable, with class instances being an exception

Declaring Sets

Syntax

Set Declaration

```
fruit_set = {"apple", "cherry", "orange", "apple"}
print(fruit_set)  # {'orange', 'cherry', 'apple'}
```

Set Constructor

```
fruit_set = set(("apple", "cherry", "orange", "apple")) # d
print(fruit_set) # {'orange', 'cherry', 'apple'}
```

Declare empty set

```
empty_set = set()
```

Different Values that are Considered Duplicates

Python determines whether 2 items are duplicates based on their hash value
This can lead to some unintuitive behaviors that make sets believe 2 conceptually different values are the same

1 and True

```
my_set = {True, 1} # {True}
```

0 and False

```
my_set = {0, False} # {0}
```

Floating Points and Integers

```
my_set = {1.00, 1} # {1.0}
```

Tuples with numeric elements

```
my_set = {(1, 2), (1.0, 2)} # {(1, 2)}
```

First Duplicate Inserted is Saved

When dealing with different values that are considered dupes in a set, the first one inserted is what gets saved

```
my_set1 = {True, 1} # True
my_set2 = {1, True} # 1
```

Length of Set + Looping

Length of Set

Use the len function

```
fruit_set = {"apple", "cherry", "orange", "apple"}
set_length = len(fruit_set)  # 3
```

Loop over Set

```
my_set = {"apple", "cherry", "orange", "strawberry"}
for fruit in my_set:
    print(fruit)
```

Access Set Item (you can't)

You cannot directly access a value in a set since they have no keys or indexes

- Sets are used to check if certain values are unique

- If you need to retrieve and use values in a data structure, use a dictionary or list

in keyword: Check if Value is in Set

Use the in operator to see if the set contains a specific value

```
my_set = {"apple", "cherry", "orange", "strawberry"}
cherry_in_set = "cherry" in my_set  # True
if cherry_in_set:
    print("Cherry is in the set")
```

Add 1 Set Item

Use the add() method

```
games_set = {"Jak 3", "Bleed", "Rise to Honor", "ATV Offroa
games_set.add("Ruiner")
print(games_set) # {'ATV Offroad Fury', 'Jak 3', 'Bleed', '
```

Remove Set Items

RECOMMENDATION:
Use discard - and forget the other methods

discard() : Remove Set Value that may Exist

This method removes a set value
If you try to remove a set value that is not present, nothing happens

Remove a set value that is present

```
games = {"Jak 3", "Bleed", "Rise to Honor", "ATV Offroad Fu
games.discard("ATV Offroad Fury")
print(games) # {"Jak 3", "Bleed", "Rise to Honor"}
```

Remove a set value that is not present

```
games = {"Jak 3", "Bleed", "Rise to Honor", "ATV Offroad Fu
games.discard("Call of duty") # KeyError: 'Call of duty'
print(games) # {"Jak 3", "Bleed", "Rise to Honor", "ATV Off
```

remove() : Remove Specific Value that you Know Exists

Remove a set value that is present

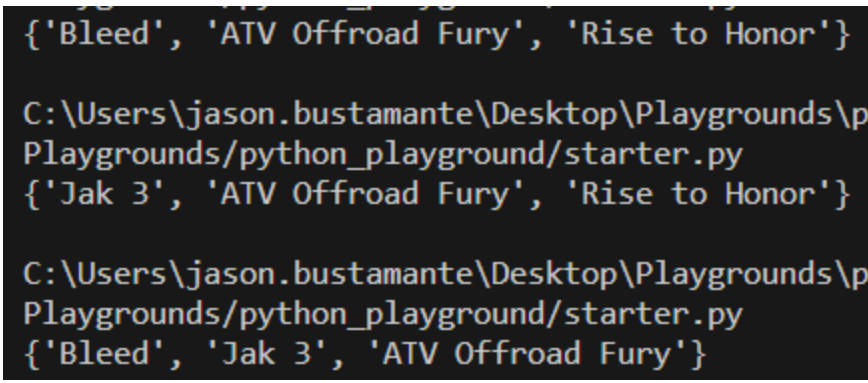
```
games = {"Jak 3", "Bleed", "Rise to Honor", "ATV Offroad Fu
games.remove("Jak 3")
print(games) # {'ATV Offroad Fury', 'Bleed', 'Rise to Honor
```

WARNING:
If you try to remove a value in a set that is not present, this produces an error

```
games = {"Jak 3", "Bleed", "Rise to Honor", "ATV Offroad Fu
games.remove("Call of duty") # KeyError: 'Call of duty'
```

pop() : Remove Random Value

```
games = {"Jak 3", "Bleed", "Rise to Honor", "ATV Offroad Fu
games.pop()
print(games) # WE RUN THE SCRIPT 3 TIMES
```



clear() : Empty the Set

```
games = {"Jak 3", "Bleed", "Rise to Honor", "ATV Offroad Fu
games.clear()
print(games) # set() (how Python logs an empty set)
```

del keyword: Delete Entire set

```
games = {"Jak 3", "Bleed", "Rise to Honor", "ATV Offroad Fu
del games
print(games) # NameError: name 'games' is not defined
```

IMPORTANT! ALL ITERABLES CAN BE CONVERTED TO SETS

Why this Matters

In the next few chapters we'll be going over methods that return info about sets

- Thanks to this lesson, these methods can be used on any iterable - strings, lists, dict keys, dict values, tuples... etc
- Excellent for solving Leetcode questions quickly without typing out much

Convert Common Iterables to Sets

Turn Strings to sets

```
my_string = "hello"
my_set = set(my_string) # {'o', 'l', 'e', 'h'}
```

Turn Lists to Sets

```
my_list = [1, 2, 2, 3, 4, 4, 5]
my_set = set(my_list) # {1, 2, 3, 4, 5}
```

Turn Dictionary Keys to sets

```
my_dict = {'a': 1, 'b': 2, 'c': 3}
key_set = set(my_dict.keys()) # {'c', 'a', 'b'}
```

Turn dictionary values to sets

```
my_dict = {'a': 1, 'b': 2, 'c': 2}
value_set = set(my_dict.values()) # {1, 2}
```

Turn Tuples to sets

```
my_tuple = (1, 2, 3, 2, 4, 5)
my_set = set(my_tuple) # {1, 2, 3, 4, 5}
```

= = = = = = = = ITERABLES

= = = = = = = =

Comparison Methods

copy() : Shallow Clone

```
games1 = {"Jak 3", "Bleed"}
games2 = games1.copy()
games2.add("Ruiner") # will not affect game1 since we add t
print(games1) # {"Jak 3", "Bleed"}
print(games2) # {'Bleed', 'Jak 3', 'Ruiner'}
```

Check if Iterables have any Values in Common

isdisjoint() Method

- Returns True if 2 sets have no values in common - and returns false otherwise
- Combine with the `not` operator to find out if 2 sets have any values in common

```
set1 = {1, 2}
set2 = {3, 4}
no_common_vals = set1.isdisjoint(set2) # True
any_common_vals = not set1.isdisjoint(set2) # False
```

Applying to Iterables other than Sets

This method works on any type of iterable if you convert them into sets first
Ex. Dictionary keys, dictionary values, lists, tuples, strings... etc

See if 2 strings have any chars in common

```
txt1 = "wail"
txt2 = "went"
charset1 = set(txt1)
charset2 = set(txt2)
chars_in_common = not charset1.isdisjoint(charset2) # not F
print(chars_in_common) # True, because we do have chars in
```

See if 2 dictionaries have any keys in common

```
dict1 = {"a": 1, "b": 2}
dict2 = {"a": 3, "d": 4} # both dictionaries have an "a" ke
key_set_1 = set(dict1.keys())
key_set_2 = set(dict2.keys())
keys_in_common = not key_set_1.isdisjoint(key_set_2) # not
print(keys_in_common) # True because we do have a key in co
```

Check if all Values in Iterable 1 are in Iterable 2

Methods

These methods can accomplish the same thing as one another if you just switch the syntax around a bit

Demo

Check if all values in `s1` are in `s2`

Approach 1: Check if `s1` is a subset of `s2`

```
s1 = {1, 2}
s2 = {1, 2, 3, 4}
is_s1_subset = s1.issubset(s2)
print(is_s1_subset) # True because s2 contains all values i
```

Approach 2: Check if `s2` is a superset of `s1`

```
s1 = {1, 2}
s2 = {1, 2, 3, 4}
is_s1_subset = s2.issuperset(s1)
print(is_s1_subset) # True because s2 contains all values i
```

Use on Other Iterables

This method may only work on sets, but can convert any iterable into a set

See if all chars in txt1 are present in txt2

```
txt1 = "man"
txt2 = "Almanac"
all_chars_present = set(txt1).issubset(set(txt2))
print(all_chars_present) # True
```

Generate Set of Unique Values

Definition: Unique Values

```
setA = {1, 2, 5}
setB = {1, 2, 9}
# unique values:    1, 2, 5, 9
# shared values:    1, 2
# unshared values:  5, 9
# Values in setA that are not in setB: 5
```

Methods: set + iterable = set of unique values

These methods take a set and any type of iterable , then produces a set of unique values between the 2

Syntax

Be aware that you can turn any iterable into a set using the set() constructor

```
unique_vals = set1.union(any_type_of_iterable)
```

union() : Concat without Mutation

Method creates a new set without mutating any set you use it on

```
utensils = {"Pencil", "Pen", "Eraser"} # not mutated
```

```
tools = {"Protractor", "Calculator"} # not mutated
all_supplies = utensils.union(tools)
print(all_supplies) # { 'Pencil', 'Pen', 'Eraser', 'Protrac
```

update() : Concat with Mutation

Mutates `set1` by adding contents of `set2`
Does not mutate `set2` in the process

```
set1.update(set2)
```

Example:

```
utensils = {"Pencil", "Pen", "Eraser"} # is mutated
extras = {"Protractor", "Calculator"} # unaffected by updat
utensils.update(extras)
print(utensils) # { 'Pencil', 'Pen', 'Eraser', 'Protractor'
```

Create Sets of Unique Values from Common Iterables

The `union` method requires at least 1 set to work
But you can use them on any pair of iterables if you convert 1 of them to sets first - like strings, lists, dictionaries, tuples... etc

Make set from...

- List + List
- Dictionary + Dictionary
- String + String

List Values

Get set of unique values between 2 lists

```
list1 = [1, 2, 3]
list2 = [1, 8, 9] # unique vals: 1, 2, 3, 8, 9
# .union must be used on a set, but its supplied argument c
set1 = set(list1) # turn list1 into set using constructor
unique_vals = set1.union(list2)
print(unique_vals) # {1, 2, 3, 8, 9} (order may vary)
```

Dictionary Keys

Get set of unique keys between 2 dictionaries

```
dict1 = { "a": 1, "b": 2, "c": 3 }
dict2 = { "x": 4, "y": 5, "c": 6 } # unique vals: a, b, c,
# .union must be used on a set, but its supplied argument c
set1 = set(dict1.keys())
unique_keys = set1.union(dict2.keys())
print(unique_keys) # {'a', 'b', 'c', 'x', 'y'} (order may v
```

Dictionary Values

Get set of unique values between 2 dictionaries

```
dict1 = { "a": 1, "b": 2, "c": 3 }
dict2 = { "a": 1, "b": 2, "c": 6 } # unique vals: 1, 2, 3,
# .union must be used on a set, but its supplied argument c
set1 = set(dict1.values())
unique_keys = set1.union(dict2.values())
print(unique_keys) # {1, 2, 3, 6} (order may vary)
```

Mix

You can perform this technique in different combinations of iterables

Set + dictionary values

```
utensils = {"Pencil", "Pen", "Eraser"} # not mutated
extras = {"key1": 5, "key2": 9} # not mutated
all_tools = utensils.union(extras.values())
print(all_tools)  # {'Pencil', 'Pen', 'Eraser', 5, 9} (orde
```

Set + Dictionary Keys

```
utensils = {"Pencil", "Pen", "Eraser"} # not mutated
extras = {"key1": 5, "key2": 9} # not mutated
all_tools = utensils.union(extras.keys())
print(all_tools)  # {'Pencil', 'Pen', 'Eraser', 'key1', 'ke
```

Set + String Chars

```
char_set = {"X", "D"} # is mutated by update()
txt = "XYZ"
unique_chars = char_set.union(txt)
print(unique_chars)  # {'X', 'D', 'Y', 'Z'} (order may vary
```

Generate Set of Shared Values

Definition: Shared Values

```
setA = {1, 2, 5}
setB = {1, 2, 9}
# unique values:    1, 2, 5, 9
# shared values:    1, 2
# unshared values:  5, 9
# Values in setA that are not in setB: 5
```

Intersection Methods

These methods produce a set of shared values between set1 and set2

Syntax

Be aware that you can turn any iterable into a set using the set() constructor

```
unique_vals = set1.intersection(set2)
```

intersection() : Get set of Shared Values with No Mutation

If you have 2 sets, use intersection() to get a new set of values they both share

```
fruits = {"apple", "banana", "cherry"} # not mutated
```

```
orgs = {"google", "microsoft", "apple"} # not mutated
shared = fruits.intersection(orgs) # returns new set
print(shared) # {'apple'} (order may vary)
```

intersection_update() : Get set of Shared Values with Mutation

This method will make an existing set equal to the shared values between it and another set

```
fruits = {"apple", "banana", "cherry"} # is mutated by meth
orgs = {"google", "microsoft", "apple"} # not mutated
fruits.intersection_update(orgs) # returns None
print(fruits) # {'apple'} (order may vary)
```

Shared Values b/t Two Iterables

The `intersection` methods can only be used on sets
But you can use them on any iterable if you convert them to sets first - like strings, lists, dictionaries, tuples... etc

List Values

Get set of shared values between 2 lists

```
nums1 = [1, 4, 9, 5]
nums2 = [6, 2, 1, 5]
# Convert lists into sets
set1 = set(nums1)
set2 = set(nums2)
shared_nums = set2.intersection(set1) # must convert lists
```

```
print(shared_nums) # {1, 5} (order may vary)
```

Dictionary Keys

Get set of shared keys between 2 dictionaries

```
dict1 = {'a': 1, 'b': 2, 'c': 3}
dict2 = {'b': 4, 'c': 5, 'd': 6}
# Get a set of keys for each dictionary first
key_set_1 = set(dict1.keys())
key_set_2 = set(dict2.keys())
shared_keys = key_set_1.intersection(key_set_2)
print(shared_keys) # {'b', 'c'} (order may vary)
```

Dictionary Values

Get set of shared values between 2 dictionaries

```
dict1 = {'a': 1, 'b': 2, 'c': 3}
dict2 = {'x': 9, 'y': 2, 'z': 3}
# Get a set of values for each dictionary first
values1 = set(dict1.values())
values2 = set(dict2.values())
shared_values = set(values1).intersection(values2)
print(shared_values) # {2, 3} (order may vary)
```

String Chars

Get list of shared chars between 2 strings

```
txt1 = "shaw"
txt2 = "hat"
# Turn each string into a set of chars first
s1 = set(txt1)
s2 = set(txt2)
shared_chars = s1.intersection(s2)
print(shared_chars) # {'h', 'a'} (order may vary)
```

Mix

You can perform this technique between any 2 iterables since we convert everything into sets before using `intersection` anyway

```
list = [1, 8, 3]
dict = { 1: "apple", 8: "banana", 9: "pear"}
# Turn each string into a set of chars first
s1 = set(list)
s2 = set(dict.keys())
shared = s1.intersection(s2)
print(shared) # {1, 8} (order may vary)
```

Generate Set of Unshared Values

Definition: Unshared Values

```
setA = {1, 2, 5}
setB = {1, 2, 9}
# unique values:    1, 2, 5, 9
# shared values:    1, 2
# unshared values:  5, 9
# Values in setA that are not in setB: 5
```

Symmetric Difference Methods

The methods produce a set of values that are unshared between set1 and set2

Syntax

Be aware that you can turn any iterable into a set using the set() constructor

```
unique_vals = set1.symmetric_difference(set2)
```

symmetric_difference() : No Mutation

"apple" is the only shared value between these 2 sets

```
fruits = {"apple", "banana", "cherry"} # not mutated
orgs = {"google", "microsoft", "apple"} # not mutated
```

```
not_shared = fruits.symmetric_difference(orgs)
print(not_shared) # {'banana', 'cherry', 'google', 'microso
```

symmetric_difference_update() : With Mutation

"apple" is the only shared value between these 2 sets

```
fruits = {"apple", "banana", "cherry"} # is mutated by meth
orgs = ["google", "microsoft", "apple"] # not mutated
fruits.symmetric_difference_update(orgs) # returns None
print(fruits) # {'banana', 'cherry', 'google', 'microsoft'}
```

Unshared Values b/t Two Iterables

The `symmetric_difference` methods can only be used on sets
But you can use them on any iterable if you convert them to sets first - like strings, lists, dictionaries, tuples... etc

List Values

Get set of unshared values between 2 lists

```
nums1 = [1, 4, 9, 5]
nums2 = [6, 2, 1, 5] # the numbers 4, 9, 6, 2 only appear i
# Convert lists into sets first
set1 = set(nums1)
set2 = set(nums2)
unshared_nums = set2.symmetric_difference(set1)
print(unshared_nums) # {2, 4, 6, 9} (these nums only appear
```

Dictionary Keys

Get set of unshared keys between 2 dictionaries

```
dict1 = {'a': 1, 'b': 2, 'c': 3}
dict2 = {'a': 4, 'x': 5, 'y': 6} # unshared keys are b, c,
# Get a set of keys for each dictionary first
key_set_1 = set(dict1.keys())
key_set_2 = set(dict2.keys())
unshared_keys = key_set_1.symmetric_difference(key_set_2)
print(unshared_keys) # {'x', 'y', 'b', 'c'}
```

Dictionary Values

Get set of unshared values between 2 dictionaries

```
dict1 = {'a': 1, 'b': 2, 'c': 3}
dict2 = {'a': 4, 'b': 2, 'c': 3} # unshared values are 1, 4
# Get a set of keys for each dictionary first
val_set_1 = set(dict1.values())
val_set_2 = set(dict2.values())
unshared_values = val_set_1.symmetric_difference(val_set_2)
print(unshared_values) # {1, 4}
```

String Chars

Get list of unshared characters between 2 strings

```
txt1 = "shaw"
txt2 = "hat" # unshared chars are s, w, t
# Turn each string into a set of chars first
char_set_1 = set(txt1)
char_set_2 = set(txt2)
unshared_chars = char_set_1.symmetric_difference(char_set_2
print(unshared_chars) # {'t', 'w', 's'}
```

Mix

You can perform this technique between any 2 iterables since we convert everything into sets before using `symmetric_difference` anyway

EXAMPLE:
Get set of unshared values between a list and a dictionary's keys

```
list = [1, 2, 3]
dict = { 1: "apple", 8: "banana", 9: "pear"}
# Turn each string into a set of chars first
s1 = set(list)
s2 = set(dict.keys())
unshared = s1.symmetric_difference(s2)
print(unshared) # {2, 3, 8, 9}
```

Generate Set of Values in Iterable 1 that are not in Iterable 2

Definition: Values in Set A that are not In Set B

```
setA = {1, 2, 5}
setB = {1, 2, 9}
# unique values:    1, 2, 5, 9
# shared values:    1, 2
# unshared values:  5, 9
# Values in setA that are not in setB: 5
```

Difference Methods

These methods produce sets of values inside set1 that are not in another iterable

Syntax

Be aware that you can turn any iterable into a set using the set() constructor

```
diff = set1.difference(any_iterable)
```

difference() : No Mutation

```
nums1 = {1, 2, 3, 4} # not mutated
nums2 = {1, 2, 8, 9} # not mutated
set1_exclusive = nums1.difference(nums2)
print(set1_exclusive) # {3, 4}
```

difference_update() : With Mutation

```
nums1 = {1, 2, 3, 4} # is mutated by method
nums2 = {1, 2, 8, 9} # not mutated
nums1.difference_update(nums2)
print(nums1) # {3, 4}
```

Values in Iterable 1 that are not in Iterable 2

The `difference` methods require at least 1 set to work
But you can use them on any pair of iterables if you convert 1 of them to sets first - like strings, lists, dictionaries, tuples... etc

List Values

Get set of values only in nums1

```
nums1 = [1, 2, 3, 4]  # 3 and 4 in nums1 but not nums2
nums2 = [1, 2, 8, 9]
set1 = set(nums1) # convert one iterable into a set first
iterable1_exclusive = set1.difference(nums2)
print(iterable1_exclusive) # {3, 4}
```

Dictionary Keys

Get set of keys only in dict1

```
dict1 = {'a': 1, 'b': 2, 'c': 3} # b and c in dict1 but not
dict2 = {'a': 4, 'x': 5, 'y': 6}
key_set_1 = set(dict1.keys())  # convert one iterable into
iterable1_exclusive = key_set_1.difference(dict2.keys())
print(iterable1_exclusive) # {'b', 'c'}
```

Dictionary Values

Get set of values only in dict1

```
dict1 = {'a': 1, 'b': 2, 'c': 3} # 2 and 3 in dict1 but not
dict2 = {'a': 1, 'x': 9, 'y': 5}
values_set_1 = set(dict1.values())  # convert one iterable
iterable1_exclusive = values_set_1.difference(dict2.values(
print(iterable1_exclusive) # {2, 3}
```

String Chars

Get set of chars in txt1 but not txt2

```
txt1 = "shaw" # s and w in txt1 but not txt2
txt2 = "hat"
char_set_1 = set(txt1) # convert one iterable into a set fi
iterable1_exclusive = char_set_1.difference(txt2)
print(iterable1_exclusive) # {'w', 's'}
```

Mix

You can perform this technique between any 2 iterables since we convert

everything into sets before using `symmetric_difference` anyway

EXAMPLE: Get set of values in list that are not keys in dict

```
list = [1, 2, 3]
dict = { 1: "apple", 8: "banana", 9: "pear"}
# Turn each string into a set of chars first
s1 = set(list)
iterable1_exclusive = s1.difference(dict.keys())
print(iterable1_exclusive) # {2, 3}
```

= = = = = = = = = TUPLES

= = = = = = = = =

Tuples: Ordered Unchangeable List that may have Duplicates

Declare + Get Length

Tuples Explained

Declare Tuple

Standard Syntax

```
my_tuple = ("Geoff", "Anna", "Ryan")
```

Tuple Constructor

```
my_tuple = tuple(("Geoff", "Anna", "Ryan"))
```

Declare Empty Tuple

```
empty_tuple1 = ()    # standard syntax (option 1)
empty_tuple2 = tuple()  # constructor (option 2)
```

Declare Tuple with 1 value (requires comma at end or else Python thinks it is a string)

```
my_tuple = ("Geoff",)     # standard syntax (option 1)
my_tuple = tuple(("Geoff",))  # constructor (option 2)
```

Get Tuple Length

Use the `len()` method

```
my_tuple = ("Geoff",)
tuple_length = len(my_tuple) # 1
```

Access Tuple Items

Refer to items in a tuple using positive or negative indexes

SAFEST: Bracket Notation + Ternary

To avoid errors caused by trying to access items that are out of bounds, use ternary + index

```
my_tuple = ('a', 'b')
c = my_tuple[5] if 5 < len(my_tuple) else None # None
```

Index (error prone)

Access second and the last item

```
fruits = ("apple", "banana", "cherry", "grape", "melon")
second = fruits[1] # banana
last = fruits[-1] # melon
```

WARNING
Accessing an item that's out of bounds produces an error

```
my_tuple = ('a', 'b')
c = my_tuple[5] # IndexError: tuple index out of range
```

Range

When you access a range, a new tuple will be returned

Positive Index Range

```
fruits = ("apple", "banana", "cherry", "grape", "melon")
some_fruits = fruits[1:4] # includes items at index 1, 2, 3
print(some_fruits) # ('banana', 'cherry', 'grape')
```

Negative Index Range

```
fruits = ("apple", "banana", "cherry", "grape", "melon")
some_fruits = fruits[-4: -1] # includes items at index 1, 2
print(some_fruits) # ('banana', 'cherry', 'grape')
```

Mixed Index Range

```
fruits = ("apple", "banana", "cherry", "grape", "melon")
some_fruits = fruits[1: -1] # includes items at index 1, 2,
print(some_fruits) # ('banana', 'cherry', 'grape')
```

in keyword: Check if Item is in Sets

```
fruits = ("apple", "banana", "cherry", "grape", "melon")
has_melon = "melon" in fruits # True
if has_melon:
    print("melon appears in the fruits tuple")
```

index() : Index of Value in Tuple

Returns the index of the value you feed the method

```
kitchenware = ("spoon", "fork", "ladle", "knife", "spoon")
x = kitchenware.index('fork')
print(x) # 1
```

WARNING:

Throws an error if item cannot be found

```
kitchenware = ("spoon", "fork", "ladle", "knife", "spoon")
x = kitchenware.index('napkin') # ValueError: tuple.index(x
```

count() : Number of Times an Item Appears in Tuple

Is case sensitive

```
kitchenware = ("Spoon", "Fork", "Spoon", "Ladle", "Knife")
x = kitchenware.count("Spoon") # 2
y = kitchenware.count("spoon") # 0
```

Round Brackets for Grouping vs Tuple Declaration

Tuples are defined using round brackets - and that can make it scary when you want to use brackets for grouping purposes

EXAMPLE.
Am I creating a tuple with an integer inside? Or is it still an integer?

```
answer = (1 if True else 2) # type is still integer
```

GOLDEN RULE
Python only considers an expression wrapped in round brackets a tuple if there is a comma separating multiple values
The only way to declare a tuple with 1 value is to place a comma after it

```
answer = (1 if True else 2)  # integer
not_a_tuple = (2) # integer
is_a_tuple = (2,) # tuple
is_also_tuple = tuple((2,)) # tuple
```

Loop Over Tuples

Regular For Loop

```
systems = ("N64", "3DS", "Xbox 360", "PS5")
for console in systems:
    print(console)
```

```
N64
3DS
Xbox 360
PS5
```

For Loop with Index

```
systems = ("N64", "3DS", "Xbox 360", "PS5")
for i in range(len(systems)):
    print([i, systems[i]])
```

```
[0, 'N64']
[1, '3DS']
[2, 'Xbox 360']
[3, 'PS5']
```

While Loop

```
systems = ("N64", "3DS", "Xbox 360", "PS5")
i = 0
while i < len(systems):
    print([i, systems[i]])
    i += 1
```

```
[0, 'N64']
[1, '3DS']
[2, 'Xbox 360']
[3, 'PS5']
```

Workarounds to Change Tuples

Workaround

Tuples have no built-in ways for you to add, remove or edit items
If you absolutely must go against what tuples are designed to do, we have a workaround:

1. Convert tuple into a list using the `list()` constructor
2. Make your edits - add/remove/edit items
3. Convert edited list back to a tuple using the `tuple()` constructor

Add Values

Add "laptop" to `gear_tuple`

```
gear_tuple = ("keyboard", "mouse", "chair")
# Convert tuple to list then add a new item
gear_list = list(gear_tuple)
gear_list.append("laptop")
# Convert list back into tuple
new_gear_tuple = tuple(gear_list)
print(new_gear_tuple) # ('keyboard', 'mouse', 'chair', 'lap
```

Remove Values

Remove "mouse" from `gear_tuple`

```
gear_tuple = ("keyboard", "mouse", "chair")
# Convert tuple to list then remove an item
gear_list = list(gear_tuple)
gear_list.pop(-2)
# Convert list back into tuple
new_gear_tuple = tuple(gear_list)
print(new_gear_tuple) # ('keyboard', 'chair')
```

Edit Values

Change the "chair" item in the tuple to "seat"

```
gear_tuple = ("keyboard", "mouse", "chair")
# Convert tuple to list then edit an item
gear_list = list(gear_tuple)
gear_list[-1] = "seat"
# Convert list back into tuple
new_gear_tuple = tuple(gear_list)
print(new_gear_tuple) # ('keyboard', 'mouse', 'seat')
```

Concatenate any Iterable on Tuples

Tuple + Tuple

```
counterpicks = ("Warioware", "Lylat")
standard = ("Smashville","Pokemon Stadium")
all_stages= counterpicks + standard
print(all_stages) # ('Warioware', 'Lylat', 'Smashville', 'P
```

Tuple + Any Iterable

You can add any iterable to an existing tuple
Just convert the iterable into a tuple using the tuple constructor first

```
counterpicks = ("Warioware", "Lylat")
standard = {1: "Smashville", 2: "Pokemon Stadium"}
# Concatenate the standard dictionary values onto the count
all_stages= counterpicks + tuple(standard.values())
print(all_stages) # ('Warioware', 'Lylat', 'Smashville', 'P
```

Multiply Tuples

```
stages = ("Battlefield", "Lylat") * 2
print(stages) # ('Battlefield', 'Lylat', 'Battlefield', 'Ly
```

IMPORTANT! ALL ITERABLES CAN BE CONVERTED TO TUPLES

Why this Matters

In the next few chapters we'll be going over methods that return info about tuples

- Thanks to this lesson, these methods can be used on any iterable - strings, lists, dict keys, dict values, tuples... etc
- Excellent for solving Leetcode questions quickly without typing out much

Convert Common Iterables to Tuples

Turn string to tuple

```
my_string = "hello"
my_tuple = tuple(my_string)  # ('h', 'e', 'l', 'l', 'o')
```

Turn List to tuple

```
my_list = [1, 2, 2, 3, 4, 4, 5]
my_tuple = tuple(my_list)  # (1, 2, 2, 3, 4, 4, 5)
```

Turn Dictionary keys to tuple

```
my_dict = {'a': 1, 'b': 2, 'c': 3}
key_tuple = tuple(my_dict.keys())  # ('a', 'b', 'c')
```

Turn Dictionary values to tuple

```
my_dict = {'a': 1, 'b': 2, 'c': 2}
value_tuple = tuple(my_dict.values())  # (1, 2, 2)
```